

每次不重样，带你收获最新测试技术！

SVN+Shell实现自动化部署	1
自由定制化 GraphQL API 的请求与返回	11
Python列表字典集合应用.....	32
Selenium Grid的三种场景部署.....	42
股市火爆之测试随想	59
Apifox接口测试详细使用教程	65
Oracle数据库分页查询的优化	86
如何更高效地开展单元测试	96



微信扫一扫关注我们



扫码填问卷，100%中奖

投稿邮箱：editor@51testing.com

SVN+Shell 实现自动化部署

◆ 作者：捞月球的 piggy

一、背景

测试环境更新的旧流程为本地 Windows 上安装 SVN 客户端，将更新包先下载到本地后，修改对应配置再传到 Linux 服务端，此过程为纯手工操作。当前通过直接使用 Linux 上的 SVN 客户端，不仅可以解决由人工本地中转的问题，也可以继续通过 Shell 脚本实现一键更新、替换配置、部署工作，从而提高部署效率。

另一方面，旧流程中每位测试人员在接手测试任务的时候都要开通 SVN 权限才能完成部署，现在把 SVN 客户端部署到测试环境，因为 SVN 客户端的默认密码功能，那么只要有服务端环境的登录权限，即可完成更新包的检出；以上述方式实现 SVN 权限从用户到用户组形式管理，让 SVN 权限模块化，也让权限管理流程变得更为简单。

二、实践

1. 问题点描述

1) 解决 SVN 在 Linux 环境的使用。

2) 由于我司当前普遍存在项目提测版本的路径或者文件名是新增变化的，需要解决 SVN 映射路径变更带来的冲突，以及让自动启动脚本减少因项目名或文件名改动的带来工作量。

3) Shell 脚本的断言、函数、逻辑控制、正则，文本处理上的使用。



2. 思路和方法

1) SVN 在 Linux 环境的常用方法:

- Linux 服务器上安装 SVN 客户端:

```
yum install -y subversion 或者 apt-get install subversion
```

- 输入 SVN 账号权限 (第一次使用需输入, 后续无需再输入)

```
svn --username=zhenglj --password XXX ls  
https://192.168.1.189/svn/Sddm\_PrinterOpsCloud/trunk/test
```

- 忽略证书, 保存密码

链接 SVN 的时候会报错: Server certificate verification failed: issuer is not trusted

```
svn ls https://192.168.1.189/svn/Sddm\_PrinterOpsCloud/
```

(R)ectect, accept (t)emporarily or accept (p)ermanently

他会提示你输入信息, 这时输入 **p**, 回车。这个命令是让忽略证书, 然后按提示走就行。

- 下载及映射到本地工作环境

```
svn checkout https://192.168.1.189/svn/Sddm\_PrinterOpsCloud/trunk/test/IBDPrinterMonitor/Sddm\_PrinterOpsCloud
```

--如上 https://192.168.1.189/svn/Sddm_PrinterOpsCloud/trunk/test/IBDPrinterMonitor 下的所有内容都将映射到本地/Sddm_PrinterOpsCloud 文件夹下:

- 了解具体更新内容

i) checkout 结束后, 可以通过反馈回的行头字段了解哪些被修改:

【?: 不在 svn 的控制中; M: 内容被修改; C: 发生冲突; A: 预定加入到版本库; K: 被锁定】

留意 A、M 行头的内容。



例：

A 20220414/sql

A 20220414/sql/更改礼物名称_20220414.sql

--A 代表有新增的文件，目前因为很多项目对，开发提测路径不稳定，因此每次更新都是删除历史文件，再全量下载做更新，5.1 的方法暂时不适用，只能用 5.2 查询日志方式去查，

ii) 通过查看日志方式了解更新备注

例：svn log

https://192.168.1.189/svn/Sddm_PrinterOpsCloud/trunk/test/IBDPrinterMonitor/IBDPrinterMonitor-updating.sql

----带文件查对应的日志，不带具体文件就可以获取到所有日志

iii) 如果开发提测路径稳定，可以每次进入到工作路径，通过 `svn up` 更新，同样通过反馈回的行头字段了解哪些被修改：

【?: 不在 svn 的控制中；M: 内容被修改；C: 发生冲突；A: 预定加入到版本库；K: 被锁定】

2) 当前普遍存在项目提测版本的路径或者文件名是新增变化的，需要解决 SVN 映射路径变更带来的工作量：

- 通过清理本地 .svn 文件和 `svn cleanup` 解决映射路径变更带来的冲突
- 通过部署包内容的关键字向上定位，找出全路径，使用了 `find` 和 `dirname` 命令
- 将变化的路径改为变量
- 建议开发人员固定路径提测

3) Shell 脚本的断言、函数、逻辑控制、正则，文本处理上的使用：

• 更新时需要判断当前步骤是否成功，才能进入下一步骤，避免脚本执行出错，故需要断言。本次断言用了 `if` 结合字符长度判断，文件存在等验证方式。

• 在部署场景中需要区分仅服务重启还是更新+替换配置+服务重启的一体化工作，用 `case` 和函数方法。



- 脚本中用了 for 循环批量删除过期文件。
- 文本处理上用了文本处理三剑客之一 awk ，结合正则表达式用法取文件。

3. 实践方案

3.1 将常用的路径定义为变量

```

1  #!/bin/bash
2  #前后端的工作路径
3  TOMCAT_WORK_DIR=/root/tomcat-8.5.59/webapps
4  PRINT_CLOUD_DIR=/root/printer_cloud_test/printer
5  TOMCAT_DAEMON=${TOMCAT_WORK_DIR}/../bin
6  JAR_NAME='printerCloud'
7  LOG_PATH=${PRINT_CLOUD_DIR}/logs
8  SVN_PATH='https://192.168.1.189/svn/Sddm_PrinterOpsCloud/trunk/test/IBDPrinterMonitor'

```

3.2 在自定义 Update 函数内完成工作路径上的更新包更新

a.如果路径是变化的，需要完成更新前清理 SVN 工作空间，删除历史更新包（不删除解压会提示要不要覆盖，所以要删除），全量下载 SVN 更新包。

```

Update()
{
#清理工作空间
if [ -e /Sddm_PrinterOpsCloud ]; then
cd /Sddm_PrinterOpsCloud
rm -rf .svn
svn cleanup
for i in $(dir ../Sddm_PrinterOpsCloud); do
rm -rf /Sddm_PrinterOpsCloud/$i
done
fi
#下载更新包
echo "svn co ${SVN_PATH}/../Sddm_PrinterOpsCloud"
result1=$(svn co ${SVN_PATH}/../Sddm_PrinterOpsCloud)
echo ${result1}
echo "下载成功"

```

b.如果路径固定，进入工作路径上直接 svn up 更新，把更新日志记录写入文件中。

```

Update()
{
#清理工作空间
cd ../Sddm_PrinterOpsCloud
result1=$(svn up)
if [ ! -d "$SVNLOG_PATH" ]; then
.... mkdir "$SVNLOG_PATH"
fi
echo ${result1} >> $SVNLOG_PATH/SVNUpdate$(date +%Y%m%d).log
echo "下载成功"

```



3.3 通过前、后台关键字查找 SVN 工作路径下的前台包、后台包全路径，并完成部署路径上的备份和替换。

```

25 if [ $? -eq 0 ]; then
26   qpackage=$(find ./Sddm_PrinterOpsCloud -name *AdminUI*.zip)
27   qpath=$(dirname ${qpackage})
28   cd ${qpath}
29   unzip ${qpackage}
30   echo "前端包解压完成！"
31   echo "前端解压包内容为 $(dir ${qpath})"
32 fi

41 echo "开始更新部署包！"
42 #部署前端
43 if [ -n "${qpackage}" ]; then
44   configpath=$(find ./Sddm_PrinterOpsCloud -name config.js)
45   cp ${TOMCAT_WORK_DIR}/pc_web_test/config.js ${configpath}
46   echo "替换配置文件成功"
47   mv $(dirname ${configpath}) ${TOMCAT_WORK_DIR}
48   configbasePath=$(basename ${dirname ${configpath}})
49   echo ${configbasePath}
50   echo "转移svn工作路径下的前端包完成"
51   #删除上一次的备份部署包
52   cd ${TOMCAT_WORK_DIR}
53   for i in $(ls|awk /^pc_web_test[0-9]{8}/); do
54     echo $i
55     rm -rf $i
56     echo "rm -rf $i"
57   done
58   mv ${TOMCAT_WORK_DIR}/pc_web_test ${TOMCAT_WORK_DIR}/pc_web_test$(date +%Y%m%d)
59   echo "前端备份成功"
60   mv ${TOMCAT_WORK_DIR}/${configbasePath} ${TOMCAT_WORK_DIR}/pc_web_test
61   echo "前端移动全量完成！"
62 fi

64 #部署后台
65 jarpath=$(find ./Sddm_PrinterOpsCloud/IBDPrinterMonitor-AdminServer-Standard -name ${JAR_NAME}.jar)
66 if [ -n "${jarpath}" ]; then
67   cd ${PRINT_CLOUD_DIR}
68   for i in $(ls|awk /^printerCloud[0-9]{8}/); do
69     rm -rf $i
70     echo "rm -rf $i"
71   done
72   echo "删除备份包成功！"
73   #备份
74   mv ${PRINT_CLOUD_DIR}/${JAR_NAME}.jar ${PRINT_CLOUD_DIR}/${JAR_NAME}$(date +%Y%m%d).jar
75   echo "备份成功！"
76   #替换部署包
77   mv ${jarpath} ${PRINT_CLOUD_DIR}
78   echo "转移后台目录成功"
79 fi
80 }

```



3.4 自定义 restart 函数，函数内完成前、后台服务的重启：

```

82 Restart() {
83     # 重启后台服务
84     if [[ ${JAR_NAME} ]]; then
85         echo "ps -xua |grep ${JAR_NAME}.jar |grep -v 'grep' |awk '{print $2}' |xargs kill--9,开始"
86         ps -xua |grep ${JAR_NAME}.jar |grep -v 'grep' |awk '{print $2}' |xargs kill--9
87         echo "ps -xua |grep ${JAR_NAME}.jar |grep -v 'grep' |awk '{print $2}' |xargs kill--9,结束"
88     fi
89     sleep 15
90     var=$(ps -xua |grep ${JAR_NAME}.jar |grep -v 'grep' |wc -l)
91     if [[ ${var} -eq 0 ]]; then
92         echo "${JAR_NAME} had beed killed!"
93     else
94         echo "进程总数是${var}"
95     fi
96     # 启动后台服务
97     JAVA_OPTS=""
98     -server
99     -Xms256m
100    -Xmx256m
101    -XX:+AlwaysPreTouch
102    -XX:+PrintGCDetails
103    -XX:+PrintGCTimeStamps
104    -XX:+PrintGCCause
105    -Xloggc:${LOG_PATH}/gc.log
106    -XX:+UseGCLogFileRotation
107    -XX:GCLogFileSize=100m
108    -XX:NumberOfGCLogFiles=5
109    -XX:+HeapDumpOnOutOfMemoryError
110    -XX:HeapDumpPath=${LOG_PATH}/heapdump
111    -Dfile.encoding=utf-8
112    cd ${PRINT_CLOUD_DIR}
113    nohup java ${JAVA_OPTS} -jar ${JAR_NAME}.jar > ${LOG_PATH}/${JAR_NAME}$(date +%Y%m%d).log.2>&1 &
114    echo "${JAR_NAME}.jar 正在启动."
115    sleep 15
116    echo -e $!

    result=$(ps -xua | grep ${JAR_NAME}.jar |grep -v 'grep' | awk '{print NR}' |wc -l)
    if [[ ${result} -eq 1 ]]; then
        echo "测试环境${JAR_NAME}服务发布成功!"
    else
        echo "测试环境${JAR_NAME}服务发布失败!"
    fi
    # 重启前端tomcat服务
    cd ${TOMCAT_DAEMON}
    sh shutdown.sh
    echo "测试环境前端服务停止中!"
    sleep 10
    result1=$(ps -ef |grep 'catalina.sh start' |grep -v 'grep' |wc -l)
    if [[ ${result1} -eq 0 ]]; then
        echo ${result1}
        echo "测试环境前端服务停止成功!"
    else
        echo ${result1}
        echo "测试环境前端服务停止失败!"
    fi
    sh startup.sh
    sleep 10
    result3=$(ps -ef |grep 'catalina.sh start' |grep -v 'grep' |wc -l)
    if [[ ${result3} == 1 ]]; then
        echo "测试环境tomcat服务发布成功!"
    else
        echo "测试环境tomcat服务发布失败!"
    fi
}

```



3.5 case 识别命令行参数，以及对应参数的函数调用：

```

147 case $1 in restart)
148     ..Restart
149     ;;
150 update)
151     ...Update
152     ...Restart
153     ;;
154     *)
155     echo "Usage: $0 {update|restart}"
156     echo "注：参数update代表下载svn上的包并部署更新，restart仅重启前后台"
157     esac

```

3.6 自动执行本次版本的 sql 脚本：

```

SQL_PATH=/Sddm_PrinterOpsCloud/sql;
for sql in $(ls /Sddm_PrinterOpsCloud/sql/);do
mysql -h 172.168.2.149 -u root -p -D ibd-pmp-prod-bak -e "source $SQL_PATH$sql";
done

```

4. 实施过程

1).服务器上安装 SVN 客户端：

```
yum install -y subversion
```

2)安装解压工具：

```
yum install -y unzip
```

3)安装 dos2unix 工具：

```
yum install -y dos2unix
```

4)给脚本增加可执行权限：

```
Chmod 777 PrintCloudUpdateOrRestart.sh
```

5)执行脚本：

```
./PrintCloudUpdateOrRestart.sh update
```

```

[root@test_05_ydy_aio_02 /]# ./PrintCloudUpdateOrRestart.sh update
svn: E155007: '/Sddm_PrinterOpsCloud' is not a working copy directory
svn co https://192.168.1.189/svn/Sddm_PrinterOpsCloud/trunk/test/IBDPrinterMonitor/ /Sddm_PrinterOpsCloud
A IBDPrinterMonitor-updating.sql A IBDPrinterMonitor-AdminServer-Standard A IBDPrinterMonitor-AdminServer-Standard/printerCloud.jar A IBDPrinterMonitor-AdminServer-Standard/lib A IBDPrinterMonitor-AdminServer-Standard/lib/httpclient-4.5.13.jar A IBDPrinterMonitor-AdminServer-Standard/lib/spring-data-keyvalue-2.5.9.jar A IBDPrinterMonitor-AdminServer-Standard/lib/knife4j-spring-ui-2.0.9.jar A IBDPrinterMonitor-AdminServer-Standard/lib/udf-http-2.1.13.jar A IBDPrinterMonitor-AdminServer-Standard/lib/udf-http-2.1.13.jar

```




```
前端包解压完成!
前端解压包内容为 dist IBDPrinterMonitor-AdminUI-Standard-1.0.5.zip
开始更新部署包!
替换配置文件成功
dist
转移svn工作路径下的前端包完成
pc_web_test20220419
rm -rf pc_web_test20220419
前端备份成功
前端移动全量完成!
rm -rf printerCloud20220419.jar
删除备份包成功!
备份成功!
转移后台目录成功
```

查看更新日志，可以看到仅后台 printerCloud.jar 做了更新：

```
[root@test_05_ydy_aio_02 svnlogs]# cat SVNUpdate20220420.log
Updating '.': Restored 'IBDPrinterMonitor-AdminServer-Standard/printerCloud.jar'
At revision 1005.
```

./PrintCloudUpdateOrRestart.sh restart

```
[root@test_05_ydy_aio_02 /]# ./PrintCloudUpdateOrRestart2.sh restart
ps xualgrep printerCloud.jar|grep -v 'grep'|awk '{print }' |xargs kill -9,开始
ps xualgrep printerCloud.jar|grep -v 'grep'|awk '{print }' |xargs kill -9,结束
printerCloud had beed killed !
printerCloud.jar 正在启动.
10613
测试环境printerCloud服务发布成功!
Using CATALINA_BASE: /root/tomcat-8.5.59
Using CATALINA_HOME: /root/tomcat-8.5.59
Using CATALINA_TMPDIR: /root/tomcat-8.5.59/temp
Using JRE_HOME: /usr/local/jdk1.8.0_261
Using CLASSPATH: /root/tomcat-8.5.59/bin/bootstrap.jar:/root/tomcat-8.5.59
/bin/tomcat-juli.jar
Using CATALINA_OPTS:
测试环境前端服务停止中!
0
测试环境前端服务停止成功!
Using CATALINA_BASE: /root/tomcat-8.5.59
Using CATALINA_HOME: /root/tomcat-8.5.59
Using CATALINA_TMPDIR: /root/tomcat-8.5.59/temp
Using JRE_HOME: /usr/local/jdk1.8.0_261
Using CLASSPATH: /root/tomcat-8.5.59/bin/bootstrap.jar:/root/tomcat-8.5.59
/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
测试环境tomcat服务发布成功!
```

./PrintCloudUpdateOrRestart.sh 未带参数或参数不正确时，给出提示：

```
[root@test_05_ydy_aio_02 /]# ./PrintCloudUpdateOrRestart2.sh
Usage: ./PrintCloudUpdateOrRestart2.sh {update|restart}
注：参数update代表下载svn上的包并部署更新，restart仅重启前后台
```



三、总结

(一) 编辑 shell 注意事项:

1.在 dos 环境编辑，文件尾行是^M\$ ，而 unix 格式的文件行尾为\$ ，需要把 dos 格式的文件转换为 unix 格式的文件：dos2unix。

2.函数定义要先写 ，才能被引用，否则会报找不到该方法。

3.kill 后要适当 sleep ，有 kill 信号中，除了 9 无条件终止进程之外，其他信号都有可能被阻塞或者忽略，而且就算是信号 9，也会先被放到 pending 队列，等到下一次 cpu 调度才会生效，所以一般我们 kill 进程之后会 sleep 一会，保证进程被杀死，否则可能会出现进程未被杀死而导致重启失败。

4.常见的杀进程方法：ps -ef|grep sss |grep -v grep|awk '{print \$2}'|xargs kill -9 在无进程的情况下会报错。ps -ef|grep sss |grep -v grep|awk '{print "kill -9" \$2}'|xargs sh 则不会，但是此方法在脚本有可执行权限是会报错：/usr/bin/kill: /usr/bin/kill: cannot execute binary file,，需要进一步解决。

5.条件判断在字符长度判断 (-z , -n) 时，变量引用需要用"" 符号，其他条件判断的变量引用则不需要 “” 号。并且注意条件判断的[] 要和其他内容空格开来。

6.-a 或-e 都可以判断文件是否存在，但取反的操作判断文件不存在只能使用[[! -e \${FILE}]]

7.搜索进程用关键字时，由于日志文件也和程序名称一样，造成查看日志也会搜索出来，对进程总数的统计造成影响，最好用 pid 来，用\$!获取上一个 pid 的返回，或只能通过精准命名避免了该情况。

(二) 微服务启动注意事项:

1.当配置分离时，java -jar 后面跟着绝对路径时会出现服务启动成功，但界面访问不上的问题，实际是因为配置文件夹是根据当前路径做的获取，这种 config 文件夹提出来的方式，要 cd 到 jar 目录启动才能读取 config 配置。不然就是读取的 jar 包里面配置文件。

综上，通过 shell 结合 Linux 的 SVN 客户端 一键进行前、后端测试环境的更新，没有其他的人工操作，减少了每个项目的部署时间。



后续如果开发提测路径可以稳定，那每次迭代部署时本地工作路径可以不需清理，还能通过 svn 的 u 标识了解到更新的对象，直接对测试概况有个基础了解，另外还减少了全量下载的带宽占用，可以更为便利。

拓展学习

[1] **【Python 自动化测试学习交流群】** 学习交流

咨询：微信 atstudy-js 备注：学习群



自由定制化 GraphQL API 的请求与返回

◆作者：NA

学习目的：本例我们将利用 jetty-server，搭建一个独立的自由定制化 GraphQL API 服务器。然后通过 Postman 的请求验证 Mock GraphQL API，最终实现一个独立的并可依据当前项目的需求和团队成员的需求的 GraphQL API 模拟服务器。

应用场景：测试环境不一样，数据不统一，从而使得一些难以捉摸的 BUG 频繁出现。测试无法重现，开发工作人员修改难，验证变得复杂，另一边用户又在抱怨和催促，管理层又要求加班加点赶工，这种情况是不是很熟悉？今天我们就开发一个独立且可定制的 GraphQL API 模拟服务器，开发团队可以快速搭建和验证原型，方便进行迭代和调整。前端、后端、测试人员可以根据各自的需求场景，独立调整和使用 GraphQL API。

温馨提示：如果您要按着示例一起做，请务必配置如下工具以及学习相关的知识。用例主要是搭建独立且可定义的 GraphQL API 模拟服务器，结合 Postman 完成 GraphQL API 的验证。示例开发过程中主要是利用 REST Assured 与 testNG 进行测试调式。另外，推荐大家与我之前写的 GraphQL API 服务器的搭建一起看。（此篇文章有 GraphQL API 搭建的详细的配置、应用步骤、验证方法 -

<http://www.51testing.com/html/18/n-7802518.html?nomobile=1>），而这篇文章是对 GraphQL API 的升级应用，从应用场景，使用人员，到使用方式更广泛，更灵活。

- IDE: IntelliJ IDEA
- 语言: Java
- API : Graphql API
- 项目类型: Maven
- API 测试工具: Postman
- 开发测试: TestNG
- 开发 API 请求: REST Assured



知识重点：

- 定制化 Graphql API 服务器搭建：需求，灵活添加或修改 GraphQL 查询和数据返回结构
- Graphql API 请求：利用 PostMan 对 Graphql API 请求验证

必要条件：

- 理解 GraphQL API 查询与验证方式
- 理解如何配置与搭建 GraphQL API 模拟服务器
- Maven 项目创建与编译

知识补充：

• GraphQL 是一个开源的，面向 API 而创造出来的数据查询操作语言以及相应的运行环境。于 2012 年仍处于 Facebook 内部开发阶段，直到 2015 年才公开发布。2018 年 11 月 7 日，Facebook 将 GraphQL 项目转移到新成立的 GraphQL 基金会。

• GraphQL 相较于 REST 以及其他 web service 架构提供了一种更加高效、强大和灵活的开发 web APIs 的方式。它通过由客户端根据所需定义数据结构，同时由服务端负责返回相同数据结构的对应数据的方式避免了服务端大量冗余数据的返回。

- GraphQL 支持数据读取、写入（操作）和数据变更订阅（实时更新）。
- 2018 年 2 月 9 日 GraphQL 的部分模式定义语言（SDL）规范制定完成。

一 实现目标

1. 用户可以通过运行 .jar（本例 MockServer.jar）文件启动 Graphql API 服务器
2. .jar 文件会读取 2 个 JSON 文件 (schema.json, response.json)

A.schema.json：定义 API 请求格式和返回数据结构

B.response.json: 定义请求和响应数据

C.用户依据需求准备这 2 个 JSON 文件，并将这两个 JSON 文件放在与 JAR 相同的目录下

- 3.用户可以发送请求并验证响应数据
4. 查询与配置文档的关系图



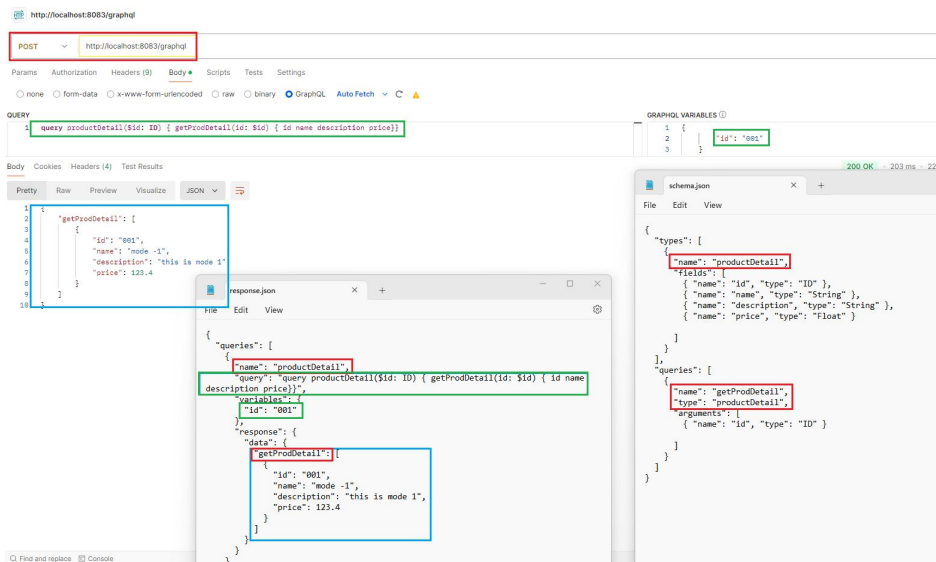
在 2 个 JSON 配置文件中用红线 框的内容 比如 name,type 是要在 2 个文件中保持一致的（大小写，空格。）

在 Postman 里请求用绿色框的内容是指在 response.json 里必须相致的，否则查无结果。

而蓝色框内的响应的数据就是 response.json 里定义的返回数据当查询语句，变量与定义相一致时。

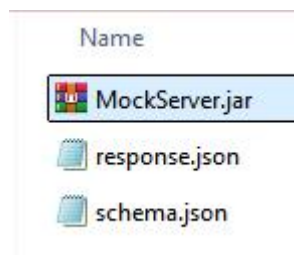
细心的朋友还会注意 schema.json 文件中 fields, argument 中的字段与类型的定义 response.json 中的 variables , getProdDetail 里的字段 时相同的。对 schema.json 就是定义了 response 的数据结构类型。

了解了这个关系图，对下面查询，准备 JSON 配置文档，以及后面的代码编写就易如反掌了。



二 目标分解

1. 我们会创建 一个项目(本示例 是 Maven 项目),此项目会依据读取 的 2 个 JSON 文档定义 API 的数据结构与响应结果， 最终会编译出一个 MockServer.jar 文件可以提供给每个人使用。



2. 2 个配置 JSON 文档，用户可以自定义文件内容，并且需要与 JAR 放置在相同的目录下 schema.json: 此文档分 2 部分，types 定义了 API 请求时 API 名称，以及需要响应的数据结构，包含响应字段名，响应字段类型。

queries 定义了 API 请求时的名称以及请求函数 "name": "getStaffVisit" 的参数结构，包含请求字段名，请求字段类型。

需要注意的是，"type": "StaffLevel" 必须与 "name": "StaffLevel" 值相应一致。

schema.json

```

schema.json
File Edit View
{
  "types": [
    {
      "name": "StaffLevel",
      "fields": [
        { "name": "department", "type": "String" },
        { "name": "name", "type": "String" },
        { "name": "visitDate", "type": "Float" },
        { "name": "level", "type": "String" },
        { "name": "confidential", "type": "Boolean" }
      ]
    }
  ],
  "queries": [
    {
      "name": "getStaffVisit",
      "type": "StaffLevel",
      "arguments": [
        { "name": "name", "type": "String" },
        { "name": "confidential", "type": "Boolean" },
        { "name": "fileName", "type": "String" }
      ]
    }
  ]
}

```

response.json 此文件包含了一个 "queries" 数组，针对 schema.json 文件中定义的 API 查询 StaffLevel 定义了 2 个响应数据依据请求时的查询条件。其中一个请求的用户 "name": "jun"，另一个请求的用户是 "name": "Mary"。

需要注意的是：

- 1) "name": "StaffLevel", 必须与 schema.json 文件中定义的 API 请求名称一致。
- 2) query 是具体的查询请求，StaffLevel 是 GraphQL API 请求名称，必须与 schema.json 和 response.json 保持一致。getStaffVisit 这个函数被定义在 schema.json 里，所以这个名称要与 schema.json 相一致。



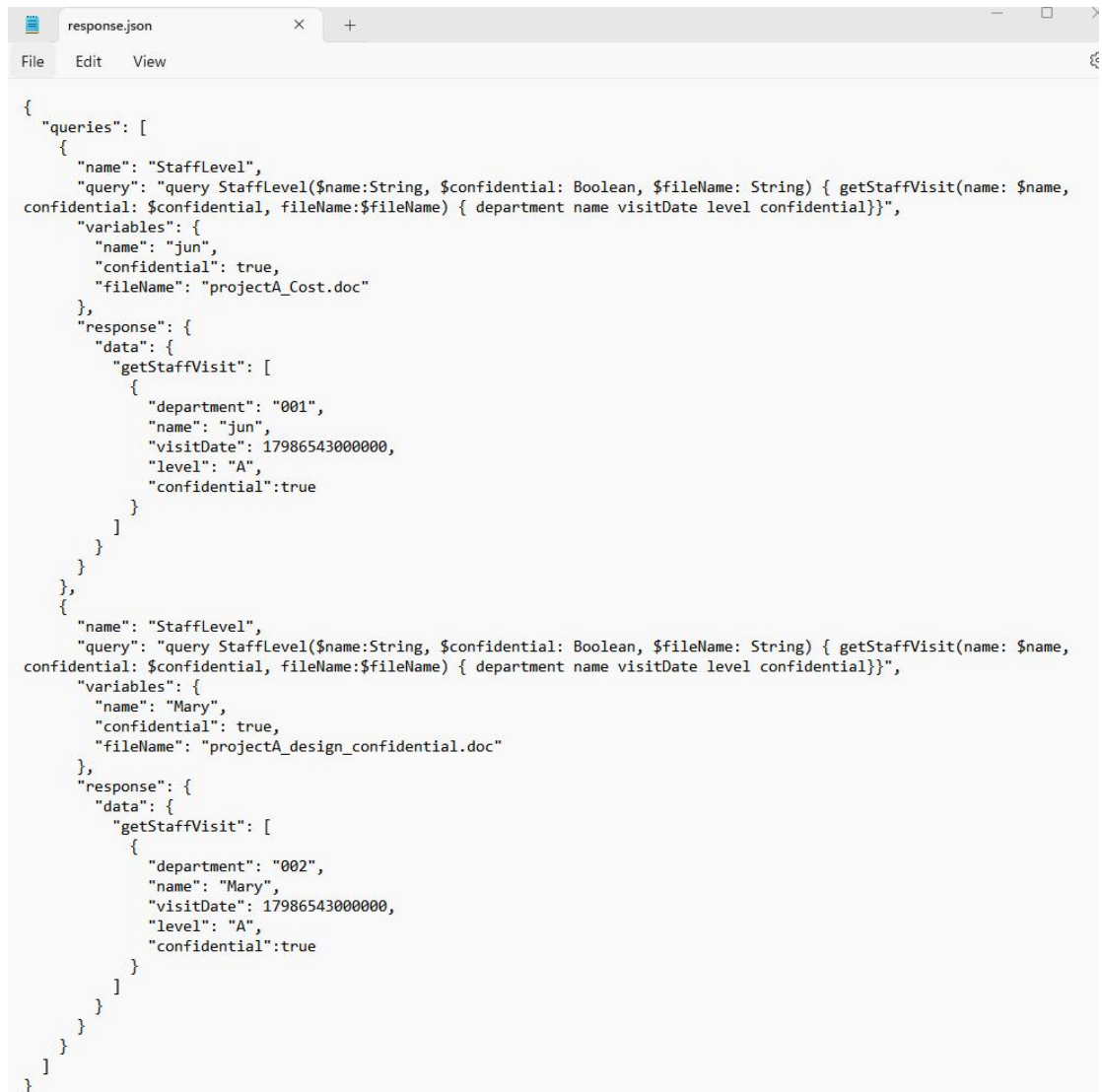
```
"query": "query StaffLevel($name:String, $confidential: Boolean, $fileName: String)
{ getStaffVisit(name: $name, confidential: $confidential, fileName:$fileName) { department name visitDate
level confidential}}",
```

3) 用户请求 API 时的数据格式会与此 query 语句进行精确的验证。

因此，用户无论是通过何种方式发送 API 请求时都需要注意语句中的空格，标点符合。否则会看到无查询记录因为查询格式不统一。(最简单的方式，就是直接复制这个 query 语句当发送 API 请求时。)

4) 函数名称 getStaffVisit 与 schema.json 以及 query 语句要保持统一。

response.json



```
{
  "queries": [
    {
      "name": "StaffLevel",
      "query": "query StaffLevel($name:String, $confidential: Boolean, $fileName: String) { getStaffVisit(name: $name, confidential: $confidential, fileName:$fileName) { department name visitDate level confidential}}",
      "variables": {
        "name": "jun",
        "confidential": true,
        "fileName": "projectA_Cost.doc"
      },
      "response": {
        "data": {
          "getStaffVisit": [
            {
              "department": "001",
              "name": "jun",
              "visitDate": 1798654300000,
              "level": "A",
              "confidential": true
            }
          ]
        }
      }
    },
    {
      "name": "StaffLevel",
      "query": "query StaffLevel($name:String, $confidential: Boolean, $fileName: String) { getStaffVisit(name: $name, confidential: $confidential, fileName:$fileName) { department name visitDate level confidential}}",
      "variables": {
        "name": "Mary",
        "confidential": true,
        "fileName": "projectA_design_confidential.doc"
      },
      "response": {
        "data": {
          "getStaffVisit": [
            {
              "department": "002",
              "name": "Mary",
              "visitDate": 1798654300000,
              "level": "A",
              "confidential": true
            }
          ]
        }
      }
    }
  ]
}
```

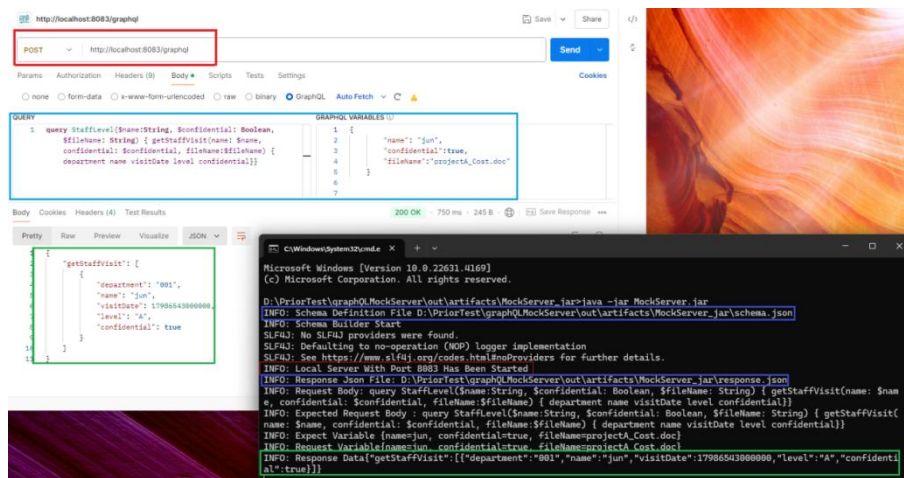


3.运行与验证：如下图所示，当执行 `java -jar MockServer.jar` 命令以后，终端会显示相应的本地服务器和相应的口已启动：`http://localhost:8083/graphql`

现在，可以通过 Postman 发送 API 查询请求，如前所述，可以直接复制查询语句从 `response.json` 文件里，然后再输入相应的参数与变量。点击查询就可以得到响应结果了。

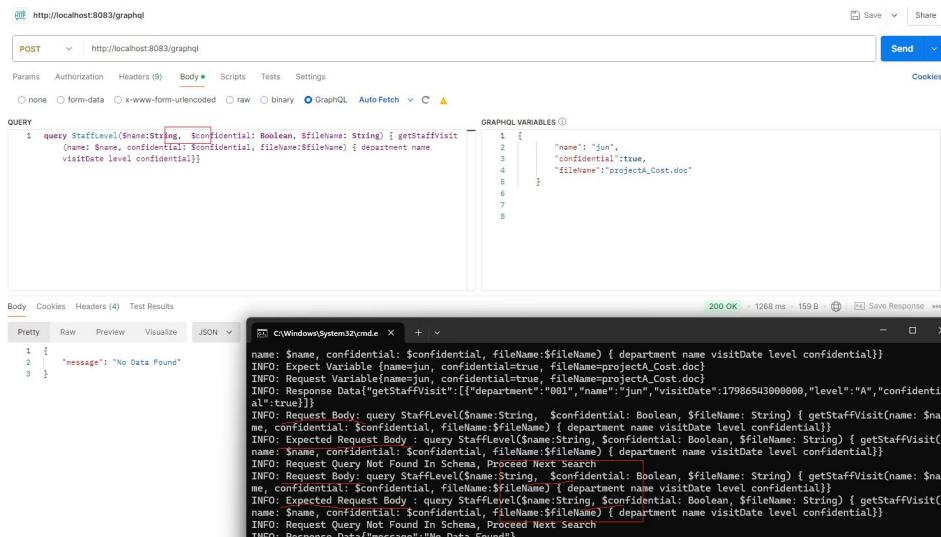
同时，在运行终端中，您会看到服务器的日志信息，记录了接收到的请求和响应数据以及配置文件的读取路径等日志信息。（这些都可以根据使用要求在开发代码的添加、优化）

示例 1：正向数据响应结果



示例 2：负向请求无响应记录

这次请求，只是在请求语句 `name` 与 `confidential` 之间加了一个空格，如前所述，这个请求格式在 `response.json` 里没有相匹配的，那结果就是无记录。（当然，本示例的验证是比较精确的，各位在具体的开发中也可以忽略空格的验证。）



4.自由定义：到目前为止，大家应该已经理解：

- 1) 这个模拟的 GraphQL API 服务器是独立运行的，
- 2) GraphQL API 的数据结构与响应是用户可以自己定义的。换句话说，可以随时增加，删除，修改这了个 API 包含数据结构与响应数据。

依据示例的 `schema.json` 与 `response.json`，这个服务器会根据提供的参数(员工名字、机密性、文件名)来获取员工的访问记录，包括他们的部门、名字、访问日期、级别和机密性信息。

现在，更新一下这个 `schema.json` 与 `response.json`，接下来依据新数据结构定义，来看下再请求与验证的结果。(此时服务器提供的就是对某一产品编号进行价格等信息的响应)

The screenshot displays the GraphQL Playground interface. On the left, the `schema.json` file is open, showing a schema with a `productDetail` type and a `getProdDetail` query. The `response.json` file on the right shows a query response with a `getProdDetail` array containing one object with fields `id`, `name`, `description`, and `price`.

The query editor shows the following query:

```
1 query productDetail($id: ID) { getProdDetail(id: $id) { id name description price}}
```

The variables editor shows:

```
1 {
2   | "id": "001"
3   | }
```

The response editor shows the following JSON response:

```
1 {
2   "getProdDetail": [
3     {
4       "id": "001",
5       "name": "mode -1",
6       "description": "this is mode 1",
7       "price": 123.4
8     }
9   ]
10 }
```

At the bottom, a terminal window shows the server logs, including the schema definition file path, schema builder start, and the request body and response data.



三 实现步骤

1. 新建一个 Maven 项目，配置构建 GraphQL API 服务器插件（这部分不太解的朋友，请参考之前给的文章链接。）

本示例 中会创建一个 MockServer 项目，下图显示了部份 pom 内容。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.example</groupId>
  <artifactId>MockServer</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Archetype - MockServer</name>
  <url>http://maven.apache.org</url>
```

2.项目文件结构:

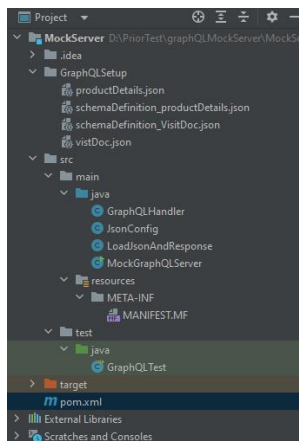
GraphQLSetup 文件夹：存放的是相对应的 schema.json 与 response.json。这里主要是为开发提供方便。(在开发中，读取 schema.json 与 response.json 就会从这个目录下，直到想要达成的功能完成时，再把读取代 JSON 文件的代码换做读取.JAR 文件的同一路径。)

src/main/java/GraphQLHandler.java: 定义 API 请求路径，处理并响应当服务器接收到 API 请求
JsonConfig.java: 处理 配置文档.json 文件的读取方式与返回对应的文件路径。

LoadJsonAndResponse.java: 处理查询请求，并依据请求 搜索响应配置 response.json，返回对应的数据结果，这个结果会被用在 GraphQLHandler.java 文件里，最终返回给用户。

MockGraphQLServer.java: 服务器启动，停止，读取 schema.json 定义 API 数据结构

src/test/java/GraphQLTest: 测试用例用来测试、调式开发的功能。



3.代码分解

MockGraphQLServer.java: main 方法中构建了服务器实例，构建 GraphQL Schema（数据结构方法）与启动端口。最后启动服务器。

```
public static void main(String[] args) {
    MockGraphQLServer graphQLServer = new MockGraphQLServer();
    GraphQLSchema schema = graphQLServer.schemaBuilder();
    int port = 8083;
    graphQLServer.serverStart(schema, port);

    // disable serverStop
    // graphQLServer.serverStop();
}
```

schemaBuilder 方法: 循环读取 2 配置 JOSN 文件中的定义，构建 API 数据结构。其中 `JsonNode rootNode = getJsonNode();`是读取 2 配置文件, `getGraphQLType(fieldType)`是根据配置文件中字段类型 转换成 `GraphQLType` 可以识别的类型，比如 json 文件类型会写成 "Boolean", 转换后是 `GraphQLBoolean`;

当前示例中可以接受的类型包含 `String,Int,Float,Boolean,ID`。

```
public GraphQLSchema schemaBuilder() {
    JsonNode rootNode = getJsonNode(); //获取根节点的 JSON 数据（即从 schema.json 文件里读取）

    // 定义类型
    Map<String, GraphQLObjectType> typeMap = new HashMap<>();
    JsonNode typesNode = rootNode.path("types");

    System.out.println("INFO: Schema Builder Start"); //开始构建 API

    //循环读取定义的 API 要返回的或者说是响应给用户的数据结构
    for (JsonNode typeNameNode : typesNode) {
        String typeName = typeNameNode.path("name").asText();
        GraphQLObjectType.Builder typeBuilder =
```



```

GraphQLObjectType.newObject().name(typeName);

        JsonNode fieldsNode = typeNode.path("fields");
        for (JsonNode fieldNode : fieldsNode) {
            String fieldName = fieldNode.path("name").asText();
            String fieldType = fieldNode.path("type").asText();
            GraphQLOutputType graphqlType = getGraphQLType(fieldType);

typeBuilder.field(GraphQLFieldDefinition.newFieldDefinition().name(fieldName).type(graphqlType));
        }
        typeMap.put(typeName, typeBuilder.build());
    }

    // 定义查询
    GraphQLObjectType.Builder queryTypeBuilder =
GraphQLObjectType.newObject().name("Query");
    JsonNode queriesNode = rootNode.path("queries");

    //循环读取定义的 API 要请求时或者是说是用户请求的数据变量结构
    for (JsonNode queryNode : queriesNode) {
        String queryName = queryNode.path("name").asText();
        String returnType = queryNode.path("type").asText();
        GraphQLObjectType returnTypeObject = typeMap.get(returnType);

        GraphQLFieldDefinition.Builder queryBuilder =
GraphQLFieldDefinition.newFieldDefinition().name(queryName).type(new
GraphQLList(returnTypeObject));
        JsonNode argumentsNode = queryNode.path("arguments");
        for (JsonNode argNode : argumentsNode) {
            String argName = argNode.path("name").asText();
            String argType = argNode.path("type").asText();
            GraphQLScalarType graphqlType = getGraphQLScalarType(argType);

queryBuilder.argument(GraphQLArgument.newArgument().name(argName).type(graphqlType));
    }
    }
  
```



```

    }

    queryTypeBuilder.field(queryBuilder.build());
}

GraphQLSchema schema = GraphQLSchema.newSchema().query(queryTypeBuilder).build();

// 返回定义的 GraphQL API 数据结构
return schema;
}

```

JsonConfig.java: 这个文件主要是处理返回需要的配置 JSON 文件的路径。这个文件有 2 个方法，其中是当没有 `schema.json` 与 `response.json` 在对应的 JAR 文件里，会读取当前项目 `GraphQLSetup` 目录下的配置 JSON 文件。而下面这个方法是在运行 JAR 时，会读取 JAR 目录下的 `schema.json` 与 `response.json`。

```

public String obtainJsonLocationJar(boolean bSchema) {
    Path jarPath;
    Path jarDirectory;
    Path jsonschemaPath;
    try {
        jarPath =
Paths.get(JsonReader.class.getProtectionDomain().getCodeSource().getLocation().toURI());
        jarDirectory = jarPath.getParent();
        String fileName;
        if (bSchema) {
            fileName = "schema.json";
        }
        else {
            fileName = "response.json";
        }
        jsonschemaPath = jarDirectory.resolve(fileName);
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }
}

```



```

    }
    String jsonPathString = jsonschemaPath.toString();
    return jsonPathString;
}

```

LoadJsonAndResponse.java 这个方法 `getResponseForQuery` 主要是接收到 `GraphQLHandler.java` 传递的请求参数(查询语句, 变量), 然后再读取配置文件 (`response.json`) 文件的内容。 第一步循环搜索验证用户的请求查询语句是否在配置文里有定义。返回查无记录, 如果有相关的定义, 那么再证请求的参数变量是否与当前请求语句的定义相一致, 如果不一致, 返回查无记录。

```

static JsonNode getResponseForQuery(String constructedQuery, Map<String, Object> requestVariables)
{
    if (queriesArray == null) {
        return createErrorResponse("INFO: Configuration not loaded"); // 当 respons.json 文
        件返回有误 时, 返回配置文档没有装载成功
    }

    // 循环查找 response.json 定义的 查询数据数组。
    for (JsonNode queryNode : queriesArray) {
        String queryText = queryNode.path("query").asText().trim();

        System.out.println("INFO: Request Body: "+ constructedQuery);
        System.out.println("INFO: Expected Request Body : "+ queryText);

        // 对比用户请求的查询是否在当前查询中的数组
        if (queryText.contentEquals(constructedQuery)) {
            JsonNode jsonVariables = queryNode.path("variables");

            // 对比是用户的请求查询中的变量是否是当前 JSON 中相一致的查询语句定义
            的相同。

            boolean variablesMatch = true;
            Map<String, Object> VariablesInJson = new LinkedHashMap<>();
            // 遍历每个查询变量并与请求变量相对比

```



```

Iterator<Map.Entry<String, JsonNode>> fields = jsonVariables.fields();
while (fields.hasNext()) {
    Map.Entry<String, JsonNode> field = fields.next();
    String key = field.getKey();
    JsonNode valueNode = field.getValue();

    if (valueNode.isTextual()) {
        VariablesInJson.put(key, valueNode.asText());
    } else if (valueNode.isNumber()) {
        // Handle different number types
        if (valueNode.isIntegralNumber()) {
            VariablesInJson.put(key, valueNode.asLong());
        } else {
            VariablesInJson.put(key, valueNode.asDouble());
        }
    } else if (valueNode.isBoolean()) {
        VariablesInJson.put(key, valueNode.asBoolean());
    } else if (valueNode.isNull()) {
        VariablesInJson.put(key, null);
    }
}

System.out.println("INFO: Expect Variable "+ VariablesInJson);
System.out.println("INFO: Request Variable"+ requestVariables);

variablesMatch = VariablesInJson.equals(requestVariables);
// 假若请求变量与定义变量相一致
// 返回 response.json 文件中的响应数据
if (variablesMatch) {
    JsonNode responseNode = queryNode.path("response").path("data");
    // Call method to get the node with key starting with "get"
    //JsonNode dynamicDataNode = getNodeStartingWithGet(responseNode);

    //JsonNode responseNode =
  
```




```

queryNode.path("response").path("data").path("getStaffVisit");

        return responseNode;
    }
    else {
        // 当查询语句中的变量与定义的语句变量不匹配时，返回变量参数没找到，继续查找
        System.out.println("INFO: Request Query Variable Not Found In Schema,
Proceed Next Search");
    }
}
else {
    // 当查询语句与定义的语句不匹配时，返回请求查询没找到，继续查找
    System.out.println("INFO: Request Query Not Found In Schema, Proceed Next
Search");
}
}
// 最终如果没有查找到用户请求，返回查无结果
return createErrorResponse("No Data Found");
}

static {
    ObjectMapper objectMapper = new ObjectMapper();
    try {
        // Load JSON configuration
        JsonConfig jsonConfig = new JsonConfig();
        String fileName = "productDetails.json"; // 设置项目运行中的 JSON 配置 (即
response.json)文件名，开发中可以灵活处理 String responseFilePath =
jsonConfig.getJsonFileFullPath(jsonConfig.obtainJsonLocationJar(false),fileName); // 这个方法会处理配
置文件的读取路径以及返回相应的路径

        System.out.println("INFO: Response Json File: " + responseFilePath);
        JsonNode rootNode = objectMapper.readTree(new File(responseFilePath));

        //获取响应数据结构(即 response.json)的根节点。
    }
}

```



// 此处是定义了一个全局变量，直接被用在了上面的 `getResponseForQuery()`方法

```

queriesArray = rootNode.path("queries");

    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

GraphQLHandler.java 此方法首先定义了请求方式(**POST**)与请求路径(`/graphql`)，接着是获取来自用户的请求包含 2 部分（查询语句，参数变量），然后把得到请求分 2 部份传递给 **LoadJsonAndResponse.java** 并得到需要返回给用户的信息，最终响应用户的请求。

`@Override`

```

public void handle(String s, Request request, HttpServletRequest httpRequest,
    HttpServletResponse httpResponse) throws IOException {
    try {
        if ("/graphql".equalsIgnoreCase(s) && "POST".equalsIgnoreCase(request.getMethod()))
        {

            // 获取用户发送的请求
            String requestBody =
request.getReader().lines().collect(Collectors.joining(System.lineSeparator()));
            // covert request boy as jsonObject
            JsonObject jsonObject = JsonParser.parseString(requestBody).getAsJsonObject();

            // 获取用户发送的请求 -查询语句
            String query = jsonObject.get("query").getString();
            //System.out.println("Request Body - query: "+ query.toString());

            /// 获取用户发送的请求 -查询变量参数
            JsonObject variablesObj = jsonObject.getAsJsonObject("variables");
            //System.out.println("Request Body - variables: "+ variablesObj.toString());

            // 重新定义查询变量参数

```



```
LinkedHashMap<String, Object> variables = new LinkedHashMap<>();
Map<String, Object> requestVariables = new LinkedHashMap<>();

variablesObj.entrySet().forEach(stringJsonElementEntry -> {
    JsonElement value = stringJsonElementEntry.getValue();
    String key = stringJsonElementEntry.getKey();

    if (value.isJsonPrimitive()){
        if (value.getAsJsonPrimitive().isBoolean()){
            requestVariables.put(key, value.getAsBoolean());
        } else if (value.getAsJsonPrimitive().isNumber()){
            requestVariables.put(key, value.getAsNumber());
        } else if (value.getAsJsonPrimitive().isString()){
            requestVariables.put(key, value.getAsString());
        }
    }
});

// 获取需要返回给用户的数据
JsonNode responseJsonData = null;
if (query.contains("get")){
    responseJsonData = LoadJsonAndResponse.getResponseForQuery(query,
requestVariables);

    System.out.println("INFO: Response Data" + responseJsonData );
}

// 重新定义返回给用户的数据结构
httpServletResponse.setContentType("application/json");
httpServletResponse.setStatus(HttpServletResponse.SC_OK);

ObjectMapper objectMapper = new ObjectMapper();
objectMapper.writeValue(httpServletResponse.getWriter(), responseJsonData);
```



```

        //objectMapper.writeValue(httpServletResponse.getWriter(),
executionResult.toSpecification());

        request.setHandled(true);
    }
    else {
        httpServletResponse.setStatus(HttpServletResponse.SC_BAD_REQUEST);
        httpServletResponse.getWriter().println("请求地址有误");
        request.setHandled(true);
    }
} catch (Exception e) {
    // 异常处理
    e.printStackTrace();

    httpServletResponse.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    httpServletResponse.getWriter().println("Internal Server Error");
    request.setHandled(true);
}
}
}

```

4.代码调试: GraphQLTest.java, 在开发中, 各种问题都可能会遇到, 所以需要一边开发一边测试自己代码的正确性。下面就是其中一个测试用例。这是基于 testNg 的框架下的测试用例, 应用 RestAssured 发出请求, 然后验证响应结果。有关 testNg 请求 GraphQL API 的方法, 请参考我附在示例中的关联文章。

@Test

```

public void test_prodDetails() {

    // 定义 graphQLServer 端口
    // 启动 graphQLServer

    MockGraphQLServer graphQLServer = new MockGraphQLServer();
    GraphQLSchema schema = graphQLServer.schemaBuilder();

    int port = 8089;

```



```

graphQLServer.serverStart(schema, port);

RestAssured.baseURI = "http://localhost:" + port + "/graphql";

// 步骤执行： 通过 RestAssured 发送 API 请求

String id = "001";

String requestBody = requestBodyProd(id);

Response response =
given().contentType("application/json").body(requestBody).log().all().post();

// 验证返回状态并打印返回结果

response.then().log().all().statusCode(200);

response.getBody().prettyPrint(); // print out response

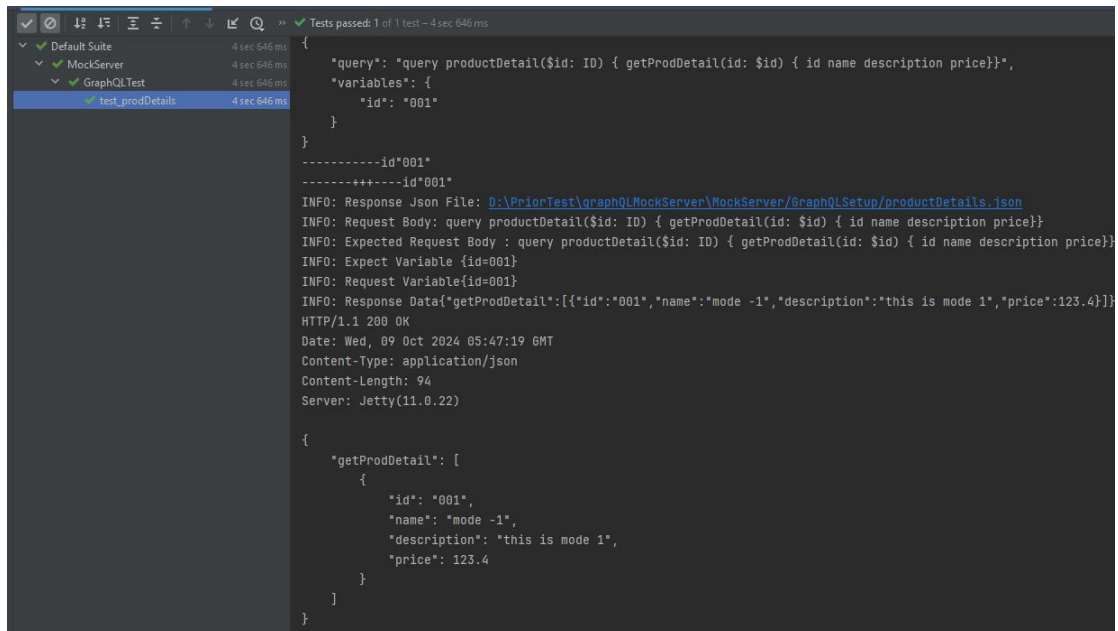
//停止服务器

graphQLServer.serverStop();

}

```

运行测试： 如果没有什么错误，运行以后你会看到类似的返回结果。



```

Tests passed: 1 of 1 test - 4sec 646ms
Default Suite 4 sec 646 ms
MockServer 4 sec 646 ms
GraphQLTest 4 sec 646 ms
test_prodDetails 4 sec 646 ms

{
  "query": "query productDetail($id: ID) { getProdDetail(id: $id) { id name description price}}",
  "variables": {
    "id": "001"
  }
}
-----id"001"
-----++++----id"001"
INFO: Response Json File: D:\PriorTest\graphql\MockServer\MockServer\graphqlSetup\productDetails.json
INFO: Request Body: query productDetail($id: ID) { getProdDetail(id: $id) { id name description price}}
INFO: Expected Request Body : query productDetail($id: ID) { getProdDetail(id: $id) { id name description price}}
INFO: Expect Variable {id=001}
INFO: Request Variable{id=001}
INFO: Response Data{"getProdDetail":[{"id":"001","name":"mode -1","description":"this is mode 1","price":123.4}]}
HTTP/1.1 200 OK
Date: Wed, 09 Oct 2024 05:47:19 GMT
Content-Type: application/json
Content-Length: 94
Server: Jetty(11.0.22)

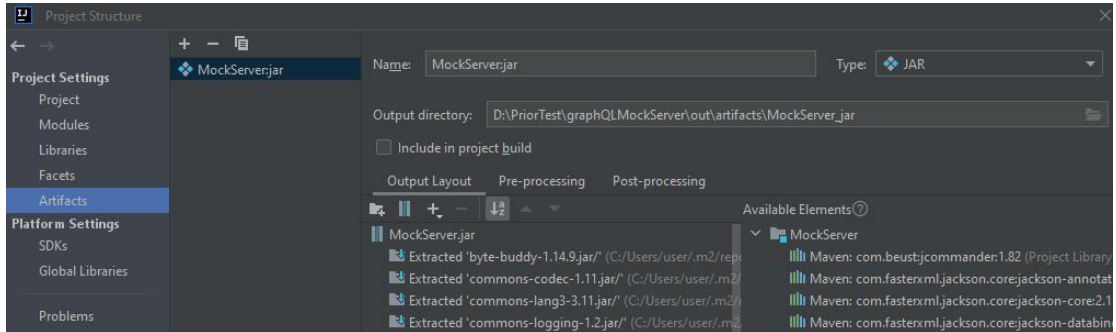
{
  "getProdDetail": [
    {
      "id": "001",
      "name": "mode -1",
      "description": "this is mode 1",
      "price": 123.4
    }
  ]
}

```

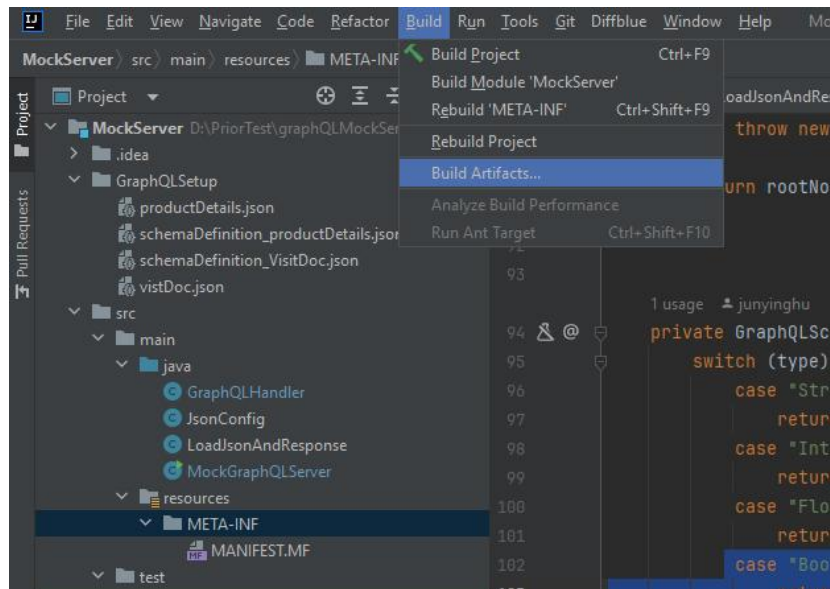


5.编译：本示例的编译步骤是用 IntelliJ IDEA 完成。

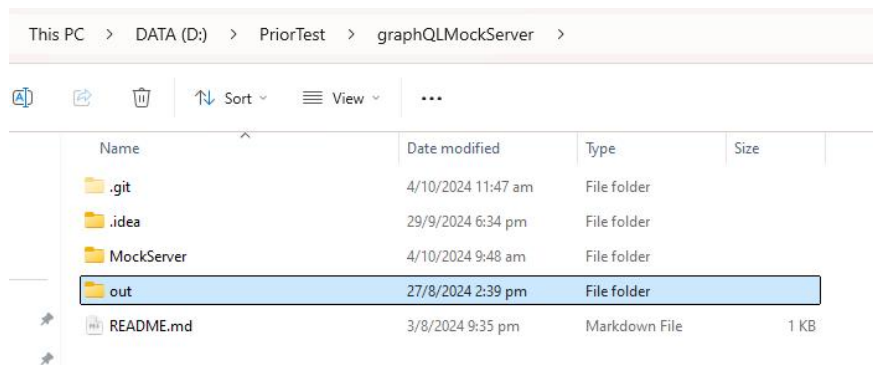
1. 在项目的 File > Project Structure >Project Settings >Artifacts 添加一个 JAR,选择当前开发的项目。



2. 添加完成以后,MANIFEST.MF 会自动生成,去到 Build>Build Artifacts > Build



编译结束以后,去到当前项目中目录,找到 out 输出目录,查看编译完成的 MockServer.jar



然后，创建 2 个 JSON 配置文件



执行 Java `-jar MockServer.jar`, 在运行控制台窗口 会显示日志 Local 服务器 端口 8083 已启动, 同时会显示 当前服务器启动读的 配置文档路径信息。

```
D:\PriorTest\graphQLMockServer\out\artifacts\MockServer_jar>java -jar MockServer.jar
INFO: Schema Definition File D:\PriorTest\graphQLMockServer\out\artifacts\MockServer_jar\schema.json
INFO: Schema Builder Start
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#noProviders for further details.
INFO: Local Server With Port 8083 Has Been Started
```

五 常见错误:

1.API 查询名称在 `schema.json` 与 `response.json` 不统一

2.API 请求时的查询与 `response.json` 定义不一致

这 2 个错误导致的结果就是响应结果一直是查无记录。直观的解决方法就是在必要的验证的步骤把 请求查询语句与参数同时与 `response.json` 定义的打印成日志里, 这样就可以快速地确认问题。

3.API 请求查询语句与参数没有分开验证。 这会导致请求的参数变量实际与 `response.json` 匹配时, 但是却有响应结果。 比如本示例的分开验证, 查询语句验证通过, 但是请求参数变量与当前的查询语句定义的不一致, 返回查无记录。这样做的好处是每个请求都能准确验证查询语句的合法性及其对应参数的一致性, 确保只有当查询语句和参数均合法时, 才会执行查询并返回结果, 适合真实的 API 在不同的业务场景中的查询与测试需求。

```
INFO: Request Body: query productDetail($id: ID) { getProdDetail(id: $id) { id name description price}}
INFO: Expected Request Body : query productDetail($id: ID) { getProdDetail(id: $id) { id name description price}}
INFO: Expect Variable {id=001}
INFO: Request Variable{id=002}
INFO: Request Query Variable Not Found In Schema, Proceed Next Search
INFO: Response Data{"message":"No Data Found"}
```



六 结语

有了这样一个既独立又可定制的 GraphQL API 模拟服务器，是不是测试会变的快捷很多，随时随地依据不同的需求返回想要的测试数据，开发环境、测试环境、用户环境、本地环境任意切换。同时，又满足自动化的测试的要求，服务器可以在每个测试用例中随时启动和关闭，确保每次测试都在一个干净且可控的环境中运行。这种灵活性不仅提高了测试效率，还使得团队能够更快速地识别和解决问题，从而提升整体开发质量和产品的可靠性。

JAR 文件下载地址：

<https://github.com/JunYinghu/graphQLMockServer/blob/main/MockServer.zip>



Python 列表字典集合应用

◆ 作者：枫叶

俗话说的好，”打铁还需自身硬“，要想让测试才能得到充分发挥，练就理论联系实践的本领是必经之路。从笔者初涉 python 语言开始，对 python 的列表、字典、集合印象就比较深刻，一是觉得他们不好区分，二是当时觉得他们有点高大上。近期对一周紧张而有序的自动化测试落地中，更是和他们进行了深层次的会话。

说起 python，不得不说它的三大数据结构：列表 Lists、字典 Dictionary 和集合 Sets。

一、列表 List

列表，又称作数组，用于按有序序列构造其他类型的数据（字符串、整数等），它使用方括号创建。

追加： 使用 `append()`方法在列表末尾添加元素

插入： 使用 `insert()`方法在指定索引处插入元素

删除： 使用 `remove()`方法删除元素

排序： 使用 `sort()`方法按升序或降序对元素进行排序

举个例子：

```
numbers = [3,6,9]
```

```
Names = [ ' Elsa' ,Lucy' , 'robot' ]
```

```
emptyList = []
```



- 1.第一行列表声明了三个整数
- 2.第二行列表存储了三个人名，按照写入顺序表示为字符串
- 3.第三行创建了空列表，有时候在编程初始化时会创建空列表，然后执行程序时再添加元素。

```
>>> numbers = [3,6,9]
>>> print(numbers)
[3, 6, 9]
>>> Names = ['Elsa','Lucy','robot']
>>> print(Names)
['Elsa', 'Lucy', 'robot']
>>> emptyList=[]
>>> print(emptyList)
[]
>>>
```

- 4.也可在列表中添加和删除元素，

```
Names.append( 'Cindy' )
```

```
Names.remove( 'Lucy' )
```

```
>>> Names.append('Cindy')
>>> Names.remove('Lucy')
>>> print(Names)
['Elsa', 'robot', 'Cindy']
>>>
```

- 5.下面小升级一下，将用列表来展示商店顾客来往情况，刚开始店内没有顾客，随之顾客的加入，可显示加入的顾客名单，随着顾客离开则及时从名单中删除，大家可以在IDE上自行练习：

```
def main():
    line = []
    action = ""
    while action != 'done':
        action = input('What action should take place?')
        if action == 'join':
            name = input('Customer name: ')
            line.append(name)
        if action == 'leave':
```



```

        name = input('Customer name: ')
        if name in line:
            line.remove(name)
        else:
            print('The person is not in line')
    if action == 'show':
        input(line)
main()

```

来看看程序的执行结果：

```

D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listttest.py
What action should take place?join
Customer name: lily
What action should take place?join
Customer name: lucy
What action should take place?join
Customer name: robot
What action should take place?show
['lily', 'lucy', 'robot']
What action should take place?leave
Customer name: lily
What action should take place?show
['lucy', 'robot']
What action should take place?leave
Customer name: blablbla
The person is not in line
What action should take place?

```

6.使用列表进行迭代，来看个例子：

```
numbers = [3,6,9,12,15]
```

```
# 输出所有数值
```

```
for i in range(len(numbers)):
```

```
print(numbers[i])
```

```

D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/listIterationtest.py
3
6
9
12
15

Process finished with exit code 0

```



7.二维列表，更能模拟真实场景来表达复杂的数据信息。如老师需要算出一个班级的学生成绩，

```
score = [
    ['Lily', 80,93,91,88],
    ['Lucy', 91,95,89,90],
    ['Robot', 96, 95, 97,93]
]

def print_final_grades(score):
    for i in range(len(score)):
        score_sum = 0
        for j in range(1,5):
            score_sum += score[i][j]
        print(score[i][0], " : ",str(score_sum/4))
```

```
print_final_grades(score)
```

来看一下我们的结果一样么？

```
Run: list2Dtest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/list2Dtest.py
Lily : 88.0
Lucy : 91.25
Robot : 95.25
Process finished with exit code 0
|
```

二、字典 Dictionary

1.字典用来存储键值对的信息，通常用大括号来表示，如姓名和电话相关：

```
contacts = {'lily':'0591-87656789','robot':'022-7788659'}
print(contacts)
```



来看看运行结果：

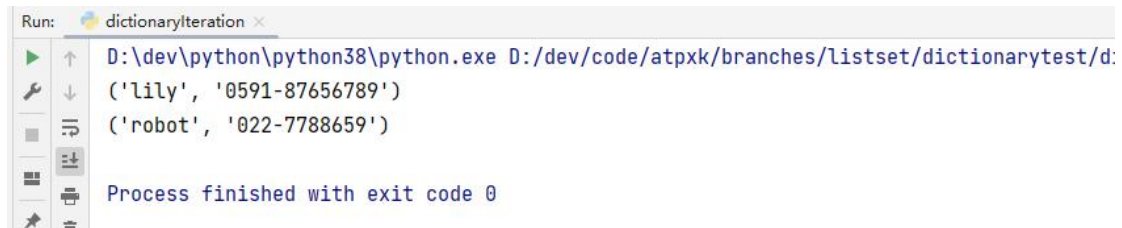


```
Run: dictionarytest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/dictionarytest/dict
{'lily': '0591-87656789', 'robot': '022-7788659'}
Process finished with exit code 0
```

2.字典的迭代，来看个例子：

```
contacts = {'lily':'0591-87656789','robot':'022-7788659'}
for contact in contacts.items():
    print(contact)
```

运行一下，得到了你们期望的吗？



```
Run: dictionaryiteration x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/dictionarytest/d:
('lily', '0591-87656789')
('robot', '022-7788659')
Process finished with exit code 0
```

三、集合 Set

1.集合可将多个元素组合在一起，而不用考虑它们的顺序。

还是上代码：

```
employee = set()
print(employee)

names = set(('lily',5,'lucy'))
print(names)

names.add('robot')
names.discard('lily')
print(names)
```



你们猜猜运行结果分别是什么？

```
Run: settest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/set/settest.py
set()
{'lucy', 5, 'lily'}
{'lucy', 5, 'robot'}
Process finished with exit code 0
```

2.集合也可以进行迭代

```
names = set(('lily',5,'lucy'))
```

```
for name in names:
```

```
    print(name)
```

```
names.add('robot')
```

```
names.discard('lily')
```

```
for name in names:
```

```
    print(name)
```

一起来看看运行结果吧

```
Run: settest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/set/settest.py
lily
lucy
5
lucy
robot
5
Process finished with exit code 0
```

3.元组 Tuple

元组与列表类似，不同之处在于元组中的元素不能进行修改。

来举个例子：

```
tuple = ('lily','lucy','robot')
```

```
print(tuple)
```



试一下修改会怎么样？

```
tuple.add('elsa')
```

```
print(tuple)
```

```
tuple.append('elsa')
```

```
print(tuple)
```



```
Run: tupletest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/tuple/tupletest.py
('lily', 'lucy', 'robot')
Traceback (most recent call last):
  File "D:/dev/code/atpxk/branches/listset/tuple/tupletest.py", line 4, in <module>
    tuple.add('elsa')
AttributeError: 'tuple' object has no attribute 'add'

Process finished with exit code 1
```



```
Run: tupletest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/tuple/tupletest.py
Traceback (most recent call last):
  File "D:/dev/code/atpxk/branches/listset/tuple/tupletest.py", line 7, in <module>
    tuple.append('elsa')
AttributeError: 'tuple' object has no attribute 'append'
('lily', 'lucy', 'robot')

Process finished with exit code 1
```

不过，我们可以对 tuple 进行拼接，前提是 tuple 的元素个数一样，比如

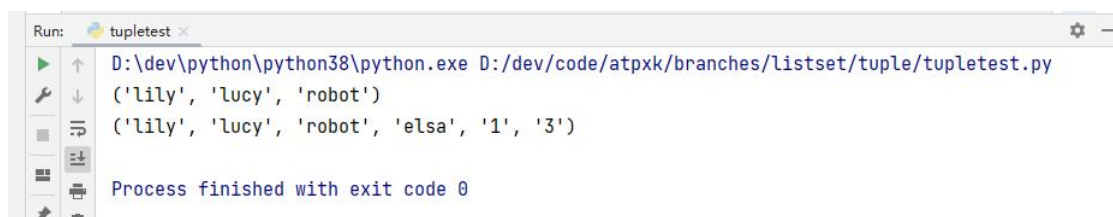
```
tuple1 = ('lily','lucy','robot')
```

```
print(tuple1)
```

```
tuple2 = ('elsa','1','3')
```

```
tuple3 = tuple1 + tuple2
```

```
print(tuple3)
```



```
Run: tupletest x
D:\dev\python\python38\python.exe D:/dev/code/atpxk/branches/listset/tuple/tupletest.py
('lily', 'lucy', 'robot')
('lily', 'lucy', 'robot', 'elsa', '1', '3')

Process finished with exit code 0
```



四、小结

集合使用小括号(), 而列表使用方括号[], 字典则用花括号{}表示, 元组用小括号()。

增加集合元素用 **add**, 删除集合元素用 **discard**, 增加数组元素用 **append**, 删除数组元素用 **remove**, 连接集合 **set3=set1+set2**。

那么元组与列表有什么区别呢? 唯一的区别就是: 列表是可修改的, 而元组是不可修改的。

五、小项目实战下

```
def test_StatisticsData(self):
    """汇总表数据"""
    url = hostname+'/collect?date=2024-07&userIds=134&t=1721812753'

    response = requests.get(url=url, headers=headers, verify=False)
    print(response.text)

    data_response_dict = response.json()
    data = json.dumps(data_response_dict, indent=2)

    # 将 JSON 字符串转换为字典
    data_object = json.loads(data)

    # 提取 "businessTotal" 值
    for item in data_object['data']['body']:
        chuqinxiaoji_value = item.get('businessTotal')
        zaigang_value = item.get('stay')
        chuchai_value = item.get('business1')
        chuguo_value = item.get('business3')

    print(chuqinxiaoji_value)
```




```

staysum_value = zaigang_value + chuchai_value + chuguo_value
print(staysum_value)

staysum_float = float(staysum_value)
staysum_fixed = format(staysum_float, '.1f')

sumValueString = staysum_fixed + '天'
print(sumValueString)

try:
    assert(sumValueString==chuqinxiaoji_value),'statistic result error!'
except AssertionError as errorMsg:
    print(errorMsg)
else:
    print('Test pass!')

def test_StatisticsTypeData(self):
    """统计表数据"""
    url = hostname+'statisticsByType?start=2023-08&end=2024-07'

    response = requests.get(url=url, headers=headers,verify=False)
    print(“统计表数据响应”,response.text)

```

一起看输出结果：

OK

```

{"msg":"操作成功","data":{"header":[{"headerName":"部门",
,"year":0,"month":0,"day":0,"type":null,"name":"depart"},{"headerName":"出差",
,"year":0,"month":0,"day":0,"type":"1","name":"_1"},{"headerName":"请假",
,"year":0,"month":0,"day":0,"type":"2","name":"_2"},{"headerName":"出国",
,"year":0,"month":0,"day":0,"type":"3","name":"_3"},{"headerName":"合计",
,"year":0,"month":0,"day":0,"type":null,"name":"total"}],"body":[{"_1":1,"total":2,"_2":1,"_3":0,"depart":"
办公室"}]},"status":200}

```

Test Over



以上主要为了让大家看实际应用，稍加解释一下，

1.加上 `print(type(response.text))`得到结果：

```
<class 'str'>
```

2.把 `string` 类型转换为 `json`

```
data_response_dict = response.json()
```

```
data = json.dumps(data_response_dict, indent=2)
```

3.将 `JSON` 字符串转换为字典

```
data_object = json.loads(data)
```

```
print(object)
```

```
print(type(object))
```

4.提取 `"sum"` 值

```
for item in data_object['data']['body']:
```

```
sum_value = item.get('sum')
```

```
    print(sum_value)
```

```
assert '2' in sum_value
```

拓展学习

[2] **【Python 自动化测试学习交流群】** 学习交流

咨询：微信 `atstudy-js` 备注：学习群

[3] ISTQB 认证基础级培训（含考试）

<http://testing51.mikecrm.com/BPY6sJs>



Selenium Grid 的三种场景部署

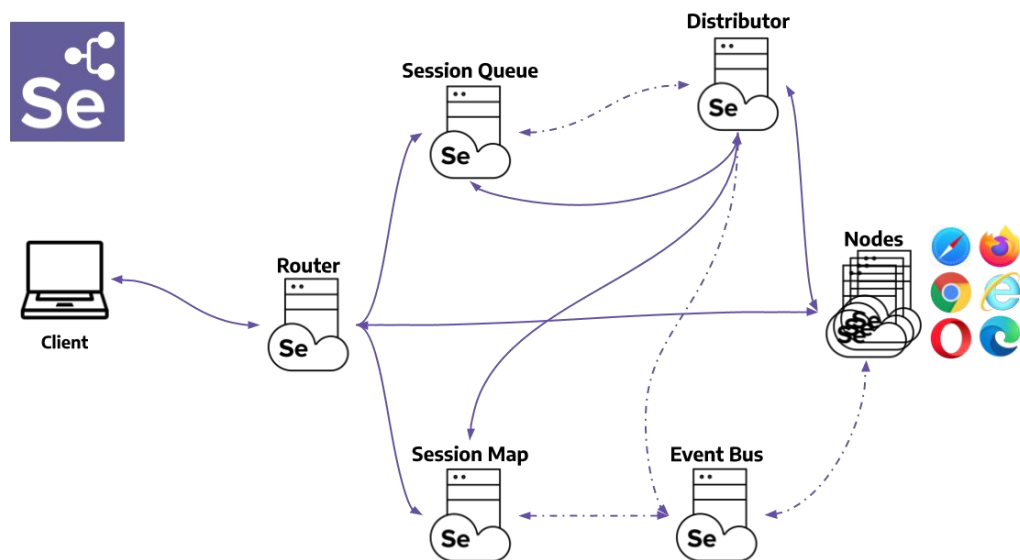
◆作者：Tynam

Selenium Grid 是 Selenium 家族中的关键一员，是一款用于分布式运行测试用例的工具。通过它，测试人员可以在不同设备、不同操作系统、不同浏览器、不同浏览器版本上执行测试用例，特别适合大规模地测试。

Selenium Grid 最新版本已经迭代到 4.x 了。下图是 Selenium Grid 4（本文后续将简称 grid）的运行原理图。图片来源 Selenium 官方文档

（<https://www.selenium.dev/documentation/grid/components/>），读者也可进入官方文档

（<https://www.selenium.dev/documentation/grid/>）了解更多内容。



grid 主要由六个组件组成：路由器（Router）、分发器（Distributor）、会话映射（Session Map）、新会话队列（New Session Queue）、节点（Node）和事件总线（Event Bus）。当测试人员（Client）向 grid 发送执行测试用例的命令后，路由器首先接收请求，如果是一个



已经存在的会话则进入会话映射查找会话运行所在的节点 ID，并将请求直接转发给节点，如果是一个新会话请求则会转发给新会话队列；新会话队列按先进先出的顺序保存所有新会话请求；分发器会轮询新会话队列查找未决的新会话请求，然后找到合适节点并创建会话。会话创建后，分发器将会话 ID 与对应节点的关系存储在会话映射。事件总线则是节点、分发器、新会话队列和会话映射之间的通信路径。Grid 的大部分内部通信都是通过消息进行的，避免了频繁发送 HTTP 请求。

了解了 grid 的原理，下面将分别介绍多设备、Docker 和 Kubernetes 环境下的部署，这三种部署方式也是技术的发展轨迹。

多设备部署

多设备部署是在不同设备上通过 Java 命令配置 Grid 工具，这也是最早的一种部署方式。

1、环境准备

Selenium Grid 是基于 Java 开发的，所以每个设备节点上都需要安装 Java 并配置环境变量，在此需要注意 Java 版本与 Grid 版本相匹配。

进入官方下载网站(<https://www.selenium.dev/downloads/>)下载 selenium-server 的 jar 包，并上传至每个设备节点。

获取所有节点 IP 信息，并规划好 Hub 节点和 Node 节点。一台设备可以同时为 Hub 节点和多个 Node 节点。例如笔者准备的设备信息如下：

设备 IP	角色	端口
192.168.10.8	Hub	4444
192.168.10.8	Node	5555

2、配置 Hub 节点

在命令行工具中输入如下命令启动 Hub 节点：

```
java -jar selenium-server-xxx.jar hub -p 4444
```



selenium-server-xxx.jar 为下载的 jar 包文件。

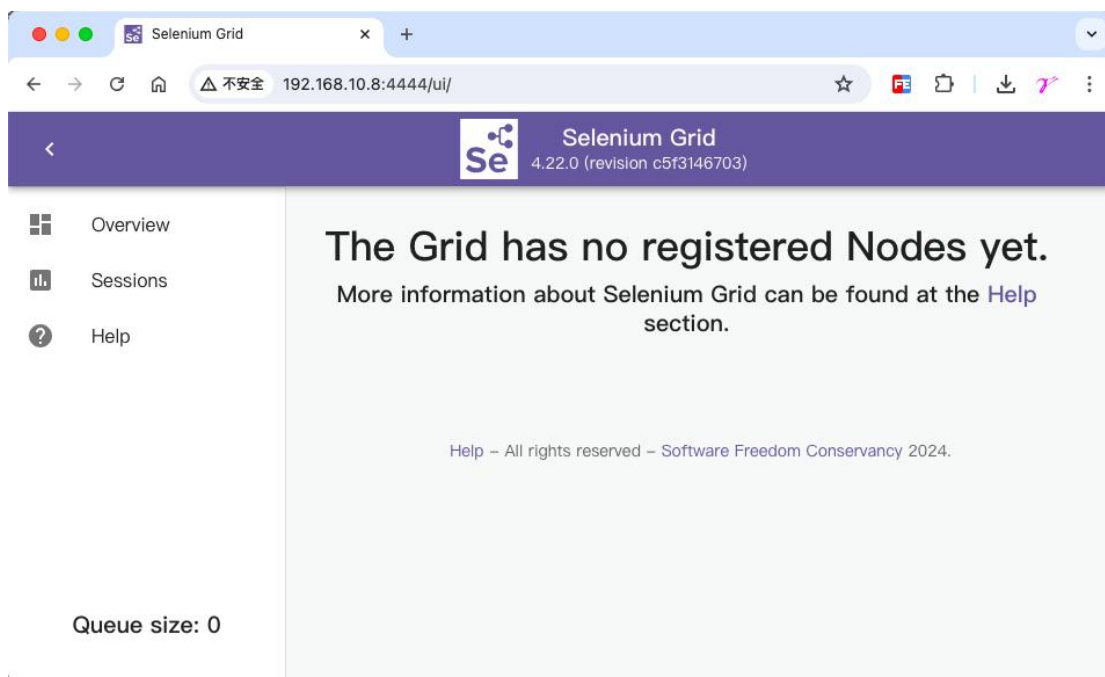
hub 表示启动的角色 hub。

-p 4444 表示指定端口为 4444，4444 为默认端口，此参数可忽略不写。

执行截图如下所示：

```
192:~ ydj$ java -jar /Users/ydj/selenium-server-4.22.0.jar hub -p 4444
21:40:39.905 INFO [LoggingOptions.configureLogEncoding] - Using the system default encoding
21:40:39.927 INFO [OpenTelemetryTracer.createTracer] - Using OpenTelemetry for tracing
21:40:40.370 INFO [BoundZmqEventBus.<init>] - XPUB binding to [binding to tcp://*:4442, advertising as tcp://[fe80:0:0:0:8b0:6fc6:907:ab9b%en0]:4442], XSUB binding to [binding to tcp://*:4443, advertising as tcp://[fe80:0:0:0:8b0:6fc6:907:ab9b%en0]:4443]
21:40:40.511 INFO [UnboundZmqEventBus.<init>] - Connecting to tcp://[fe80:0:0:0:8b0:6fc6:907:ab9b%en0]:4442 and tcp://[fe80:0:0:0:8b0:6fc6:907:ab9b%en0]:4443
21:40:40.574 INFO [UnboundZmqEventBus.<init>] - Sockets created
21:40:41.583 INFO [UnboundZmqEventBus.<init>] - Event bus ready
21:40:43.421 INFO [Hub.execute] - Started Selenium Hub 4.22.0 (revision c5f3146703): http://192.168.10.8:4444
```

初始化完成后，可通过 Hub 机 IP 访问 Console 页面。访问地址：<http://{IP}:4444>，从启动的输出日志中也可以看到访问地址，如上图就是 <http://192.168.10.8:4444>。访问页面截图如下图所示：



3、配置 Node 节点

在命令行工具中输入如下命令注册 Node 节点：

```
java -jar /Users/ydj/selenium-server-xxx.jar node --hub http://{IP}:4444 -p 5555
```

selenium-server-xxx.jar 为下载的 jar 包文件。

node 表示启动的角色 node。

-port 5555 表示指定端口为 5555, 5555 为默认端口，此参数可忽略不写。

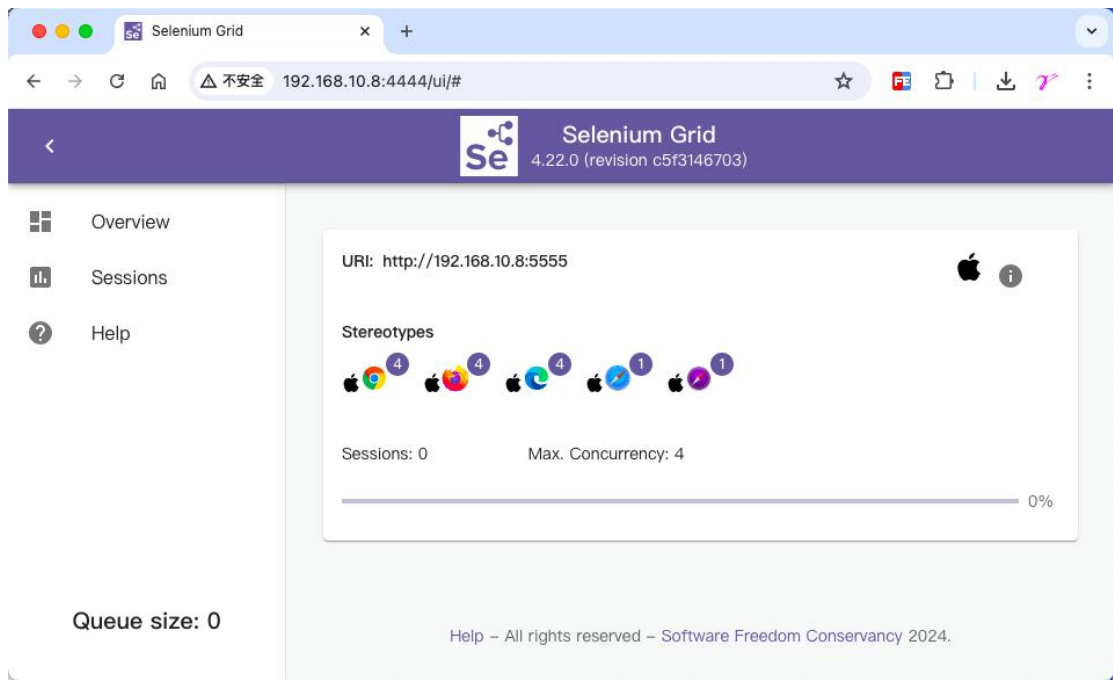
-hub http://{IP}:4444 用于指定注册的 Hub 地址，如果 Hub 节点和 Node 节点在同一设备上，此参数可忽略不写。

执行截图如下所示：

```
192:~ ydj$ java -jar /Users/ydj/selenium-server-4.22.0.jar node --hub http://192.168.10.8:4444 -p 5555
21:57:50.226 INFO [LoggingOptions.configureLogEncoding] - Using the system default encoding
21:57:50.234 INFO [OpenTelemetryTracer.createTracer] - Using OpenTelemetry for tracing
21:57:50.434 INFO [UnboundZmqEventBus.<init>] - Connecting to tcp://192.168.10.8:4442 and tcp://192.168.10.8:4443
21:57:50.495 INFO [UnboundZmqEventBus.<init>] - Sockets created
21:57:51.502 INFO [UnboundZmqEventBus.<init>] - Event bus ready
21:57:51.741 INFO [NodeServer.createHandlers] - Reporting self as: http://192.168.10.8:5555
21:57:51.766 INFO [NodeOptions.getSessionFactories] - Detected 4 available processors
21:57:51.769 INFO [NodeOptions.discoverDrivers] - Looking for existing drivers on the PATH.
21:57:51.770 INFO [NodeOptions.discoverDrivers] - Add '--selenium-manager true' to the startup command to setup drivers automatically.
21:57:52.065 WARN [SeleniumManager.lambda$runCommand$1] - Unable to discover proper chromedriver version in offline mode
21:57:52.352 WARN [SeleniumManager.lambda$runCommand$1] - Unable to discover proper msedgedriver version in offline mode
21:57:52.414 WARN [SeleniumManager.lambda$runCommand$1] - Unable to discover proper geckodriver version in offline mode
21:57:52.548 WARN [SeleniumManager.lambda$runCommand$1] - Unable to download safaridriver in offline mode
21:57:52.595 INFO [NodeOptions.report] - Adding Safari Technology Preview for {"browserName": "Safari Technology Preview", "platformName": "mac"} 1 times
21:57:52.597 INFO [NodeOptions.report] - Adding Chrome for {"browserName": "chrome", "platformName": "mac"} 4 times
21:57:52.599 INFO [NodeOptions.report] - Adding Safari for {"browserName": "safari", "platformName": "mac"} 1 times
21:57:52.601 INFO [NodeOptions.report] - Adding Edge for {"browserName": "MicrosoftEdge", "platformName": "mac"} 4 times
21:57:52.607 INFO [NodeOptions.report] - Adding Firefox for {"browserName": "firefox", "platformName": "mac"} 4 times
21:57:52.697 INFO [Node.<init>] - Binding additional locator mechanisms: relative
21:57:53.035 INFO [NodeServer$1.start] - Starting registration process for Node http://192.168.10.8:5555
21:57:53.037 INFO [NodeServer.execute] - Started Selenium node 4.22.0 (revision c5f3146703): http://192.168.10.8:5555
21:57:53.075 INFO [NodeServer$1.lambda$start$1] - Sending registration event...
21:57:53.269 INFO [NodeServer.lambda$createHandlers$2] - Node has been added
```

当出现“Node has been added”的提示就表示 Node 节点注册成功。再次访问 Hub 地址 http://{IP}:4444 可以看到已经成功注册一个 Node 节点，如下截图所示：





通过上面命令可继续注册 Node 节点。

Docker 部署

docker 是一个开源的容器化平台，可以将应用程序及其所有依赖项打包在一个独立、可移植的容器中。它通过利用容器技术来实现应用程序的快速部署、可扩展性和跨平台性。Docker 部署是一种虚拟化的部署，可以使多台设备上程序运行的环境保持一致。采用 Docker 技术比在多台设备或虚拟机上分批部署更加轻便、快速。

下面就来开始 Docker 上部署 Grid。

1、环境准备

设备机器上需要安装 Docker。

关于 Docker 的安装，基本使用可参考 51 讲堂中《学会 Docker，测试环境轻松搞定》(<http://quan.51testing.com/pcQuan/lecture/159>) 一文。

本次我们采用 docker-compose 命令运行 YAML 文件来部署。

2、下载 Grid 相关镜像。

将需要的 hub 镜像和 node 镜像下载到本地，如果本地没有对应的镜像和标签，docker-compose 运行时则会从远程仓库拉取。



使用如下命令拉取 `selenium/hub` 镜像，如果没有特殊要求，我们拉取最新版本 `latest` 镜像就行。

```
docker pull selenium/hub:latest
```

操作截图如下所示：

```
192:~ ydj$ docker pull selenium/hub:latest
latest: Pulling from selenium/hub
7646c8da3324: Pull complete
a72d41d4a694: Pulling fs layer
89baa623acb8: Download complete
c1e7cc7f308f: Download complete
5dd5f01f7d1f: Download complete
e34361cd42fc: Download complete
```

接着再使用如下命令拉取 `selenium/node-chrome` 镜像，不同浏览器有不同的 `Node` 镜像，在此以 `Chrome` 为例。

```
docker pull selenium/node-chrome:latest
```

`hub` 和 `node` 镜像都获取成功后，使用命令 `docker images selenium/*` 查看本地所有 `selenium` 相关的镜像，截图如下：

```
192:~ ydj$ docker images selenium/*
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
selenium/node-chrome latest       b219df0a03a8     2 years ago     1.19GB
selenium/hub        latest       d2eb364e7e29     2 years ago     366MB
192:~ ydj$
```

3、构造 `docker-compose.yaml` 文件文件

`Docker Compose` 是用于定义和运行多容器应用程序的工具，`Compose` 简化了对整个应用程序堆栈的控制，让您能够在单个易于理解的 `yaml` 配置文件中轻松管理服务、网络和卷。然后，只需一个命令，即可创建并启动所有服务 从您的配置文件中。

创建一个 `docker-compose.yaml` 文件，内容如下：

```
services:
```




```
hub:                                     # hub 容器

  image: selenium/hub:latest

  container_name: selenium-hub

  ports:

    - "4444:4444"

  networks:

    - net

chrome1:                                 # 第一个 chrome 节点容器

  image: selenium/node-chrome:latest

  depends_on:

    - hub

  environment:

    - SE_EVENT_BUS_HOST=hub

    - SE_EVENT_BUS_PORT=4444

    - SE_EVENT_BUS_PUBLISH_PORT=4442

    - SE_EVENT_BUS_SUBSCRIBE_PORT=4443

  networks:

    - net

chrome2:                                 # 第二个 chrome 节点容器

  image: selenium/node-chrome:latest

  depends_on:

    - hub

  environment:

    - SE_EVENT_BUS_HOST=hub

    - SE_EVENT_BUS_PORT=4444

    - SE_EVENT_BUS_PUBLISH_PORT=4442

    - SE_EVENT_BUS_SUBSCRIBE_PORT=4443

  networks:

    - net

networks:

  net:
```



上面文件中，我们定义了三个容器，一个 **hub** 容器和两个 **chrome** 节点容器，如果需要注册更多节点，则参考上面节点内容复制继续添加即可。

下面对上面文件中关键字段做以下说明：

depends_on: 定义容器启动顺序，上文件中 **depends_on/hub** 表示 **hub** 节点启动后再启动 **chrome** 节点。

SE_EVENT_BUS_HOST: **hub** 节点 IP 地址，在 **docker-compose** 可直接写 **hub** 名称。

SE_EVENT_BUS_PORT: **hub** 节点端口，即 Selenium Grid 控制台访问端口。

SE_EVENT_BUS_PUBLISH_PORT: **hub** 节点的事件总线发布事件的端口。

SE_EVENT_BUS_SUBSCRIBE_PORT: **hub** 节点的组件订阅事件的端口。

networks: 通常应用于集群服务，可以使不同应用程序在相同的网络中运行。

注意：**Hub** 节点在启动时，如果没有指定控制台访问端口、总线发布事件的端口、订阅事件的端口，则默认为 4444、4442、4443。

4、启动容器

使用如下命令启动 **docker-compose.yml** 中的服务。

```
docker-compose up -d
```

docker-compose up -d 命令是用来启动 Docker Compose 项目中定义的服务的，并且让这些服务在后台以守护进程（**daemon**）模式运行。

up: 根据 **compose** 文件中的定义启动或重新创建服务。

-d: 标志位参数，表示“**detached mode**”，即后台运行模式。

操作截图如下所示：



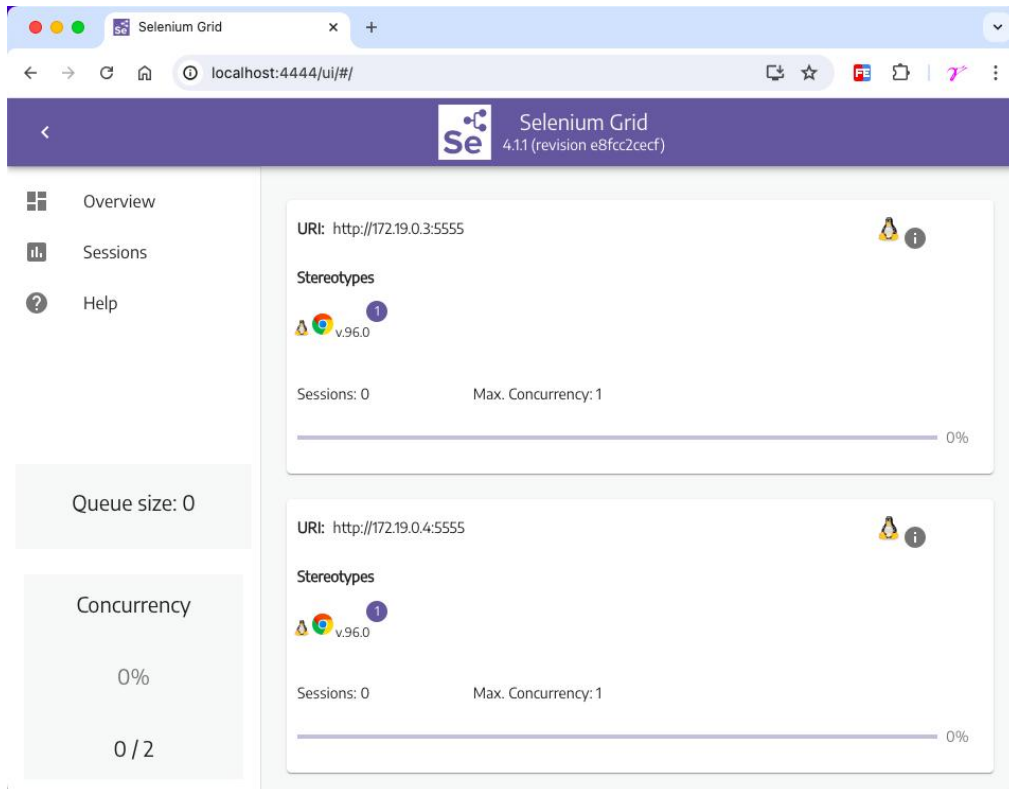
```
192:~ ydj$ docker-compose up -d
[+] Running 4/4
 ✓ Network ydj_net          Created
 ✓ Container selenium-hub   Started
 ✓ Container ydj-chrome1-1  Started
 ✓ Container ydj-chrome2-1  Started
192:~ ydj$
```

从输出日志中可以看到四条有效信息，网络创建成功，容器 `selenium-hub` 和 两个 `chrome node` 也启动成功。

在命令行工具中使用命令 `docker ps` 命令也可 查看启动的容器，如下图所示：

```
192:~ ydj$ docker ps
CONTAINER ID   IMAGE                                COMMAND
CREATED       STATUS    PORTS
e7ac5a91afd9  selenium/node-chrome:latest        "/opt/bin/entry_poin...
8 minutes ago Up 8 minutes 5900/tcp
5588d7b57b7f  selenium/node-chrome:latest        "/opt/bin/entry_poin...
8 minutes ago Up 8 minutes 5900/tcp
33f7d763bf53  selenium/hub:latest                "/opt/bin/entry_poin...
8 minutes ago Up 8 minutes 4442-4443/tcp, 0.0.0.0:4444->4444/tcp selenium-hub
```

容器创建成功后，便可在本地通过 `http://localhost:4444` 访问 `Grid` 页面。如果设置了域名也可通过域名访问。截图如下：



从截图中看到，一共注册了两个 Node 节点，节点中只有一个 Chrome 浏览器，与 docker-compose.yaml 文件中的编排相匹配。

Kubernetes 部署

kubernetes 简称 K8s，是一个开源的，用于管理云平台中多个主机上的容器化的应用，Kubernetes 的目标是让部署容器化的应用简单并且高效，Kubernetes 提供了应用部署，规划，更新，维护的一种机制。其本质是一组服务器集群，可以在集群的每个节点上运行特定的程序，来对节点中的容器进行管理。目的是实现资源管理的自动化。主要优点有：自我修复、弹性伸缩、服务发现、负载均衡、版本回退、存储编排等。

1、环境准备

需要在设备上先安装 Docker 和 Kubernetes。

Docker 安装，参考 51 讲堂中《学会 Docker，测试环境轻松搞定》
(<http://quan.51testing.com/pcQuan/lecture/159>) 一文。

Kubernetes 安装及学习可参考知乎网站《一文让你全面了解 K8s(Kubernetes)》
(<https://zhuanlan.zhihu.com/p/621593509>) 一文。

2、Grid 部署逻辑

Grid 部署需要分三步走，首先是部署 hub；然后创建一个 grid service 服务将 hub 端口映射出来，使外部可以访问；最后部署 nodes，通过 grid service 将 nodes 注册到 hub 节点。grid service 可在 hub 和 nodes 之间进行双向通信。

本次演示还是以 yaml 配置文件为例进行制作各个节点或服务。因此需要准备 hub deploy、node deploy 和 grid service 三个文件。

3、hub deploy

新建一个 hub-deploy.yaml 文件，用以配置 hub 节点容器。内容如下：

```
apiVersion: apps/v1
```



```
kind: Deployment
metadata:
  name: selenium-hub-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: selenium-hub
      role: hub
  template:
    metadata:
      labels:
        app: selenium-hub
        role: hub
    spec:
      containers:
        - name: selenium-hub
          image: selenium/hub:4.7.0
          ports:
            - containerPort: 4444
            - containerPort: 4442
            - containerPort: 4443
```

下面对上面文件中关键字段做以下说明：

kind: 指定创建资源的角色/类型，由于我们只是部署一个应用，所以选择 **Deployment**。

metadata: 资源的元数据/属性。

metadata.name: 资源的名字，在同一个 **namespace** 中必须唯一。上面文件中没有定义 **namespace**，一般取默认空间名 **default**。

template: 模版。

template.metadata: 资源的元数据/属性。



template.metadata.labels: 设定资源的标签。

spec: 指定该资源的内容。

spec.replicas: 副本数量，即创建 pod 的数量。

containers: 定义 pod 中容器内容，可以同时添加多个容器。

containers.image: 镜像。

selector: 选择器。

matchLabels: 匹配标签。

然后通过命令 `kubectl apply -f hub-deploy.yaml` 启动 hub 容器，如下图：

```

ydjdeMacBook-Air:kind ydj$ kubectl apply -f hub-deploy.yaml
deployment.apps/selenium-hub-deploy created
ydjdeMacBook-Air:kind ydj$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
selenium-hub-deploy-87f4d4c47-mmfhv 1/1     Running   0           14s
ydjdeMacBook-Air:kind ydj$

```

启动后可使用命令 `kubectl get pods` 查看已经运行的 pods，如上图，可以看到一个 `selenium-hub-deploy-*` 的 pod 的 status 状态是 `Running`，证明刚才的 `hub deploy` 创建成功。

4、hub service

新建一个 `hub-service.yaml` 文件，用于将部署的 `hub` 容器端口映射出来。文件内容如下：

```

apiVersion: v1
kind: Service
metadata:
  name: selenium-grid-service
spec:
  type: NodePort
  selector:
    app: selenium-hub

```



```
ports:
  - protocol: TCP
    port: 4444
    targetPort: 4444
    nodePort: 30001
    name: port1
  - port: 4442
    targetPort: 4442
    nodePort: 30002
    protocol: TCP
    name: port2
  - port: 4443
    nodePort: 30003
    targetPort: 4443
    protocol: TCP
    name: port3
```

下面对上面文件中关键字段做以下说明：

kind: 值为 **Service**，表示启动的是一个服务。

metadata.name: 指定服务的名称，在此指定为 **selenium-grid-service**。

spec.type: 指定服务的类型，在此设置为 **NodePort**，使用 **node** 节点的 IP 加端口可以访问 Pod 服务，**master** 节点 IP 不可以访问，端口范围 30000-32767。

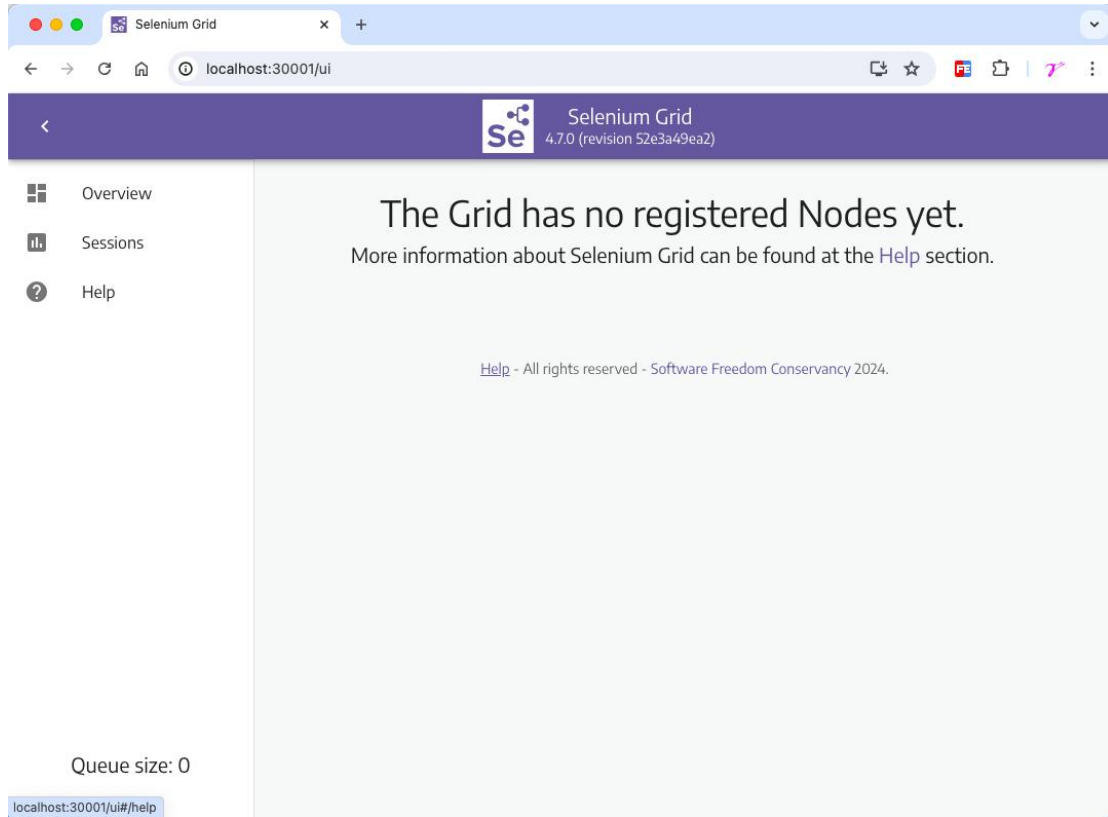
然后通过命令 **kubectl apply -f hub-service.yaml** 启动服务，如下图：

```
ydjdeMacBook-Air:kind ydj$ kubectl apply -f hub-service.yaml
service/selenium-grid-service created
ydjdeMacBook-Air:kind ydj$ kubectl get svc
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes          ClusterIP   10.96.0.1     <none>         443/TCP          13d
selenium-grid-service NodePort    10.100.166.33 <none>         4444:30001/TCP,4442:30002/TCP,4443:30003/TCP 9s
ydjdeMacBook-Air:kind ydj$
```

启动后可使用命令 **kubectl get svc** 查看已经运行的服务，如上图，可以看到一个 **selenium-grid-service** 的服务，并且映射了三个端口 4444:30001、4442:30002 和 4443:30003。



然后本地访问 `http://localhost:30001` 即可查看 `hub console` 页面。如下图所示。当然，也可通过 `Ingress` 设置域名访问。



5、node deploy

新建一个 `node-deploy.yaml` 文件，用以配置 `node` 节点容器。内容如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: selenium-node-chrome-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: selenium-node-chrome
      role: node
      browser: chrome
```




```
template:
  metadata:
    labels:
      app: selenium-node-chrome
      role: node
      browser: chrome
  spec:
    containers:
      - name: selenium-node-chrome
        image: selenium/node-chrome:4.7.0
        ports:
          - containerPort: 5555
        env:
          - name: SE_EVENT_BUS_HOST
            value: "selenium-grid-service"
          - name: SE_EVENT_BUS_PUBLISH_PORT
            value: "4442"
          - name: SE_EVENT_BUS_SUBSCRIBE_PORT
            value: "4443"
          - name: SE_NODE_GRID_URL
            value: "http://localhost:30001"
```

下面对上面文件中关键字段做以下说明：

replicas: 在此设置 3，表示创建 3 个副本，即注册三个子节点。

containers.env: 用以设置环境变量。

SE_EVENT_BUS_HOST: hub 节点 IP 地址，在 k8s 中可以直接写 grid service 名称。

SE_EVENT_BUS_PUBLISH_PORT: hub 节点的事件总线发布事件的端口。

SE_EVENT_BUS_SUBSCRIBE_PORT: hub 节点的组件订阅事件的端口。

SE_NODE_GRID_URL: Grid 公共 URL(通常是 Hub 或 Router 的地址)。

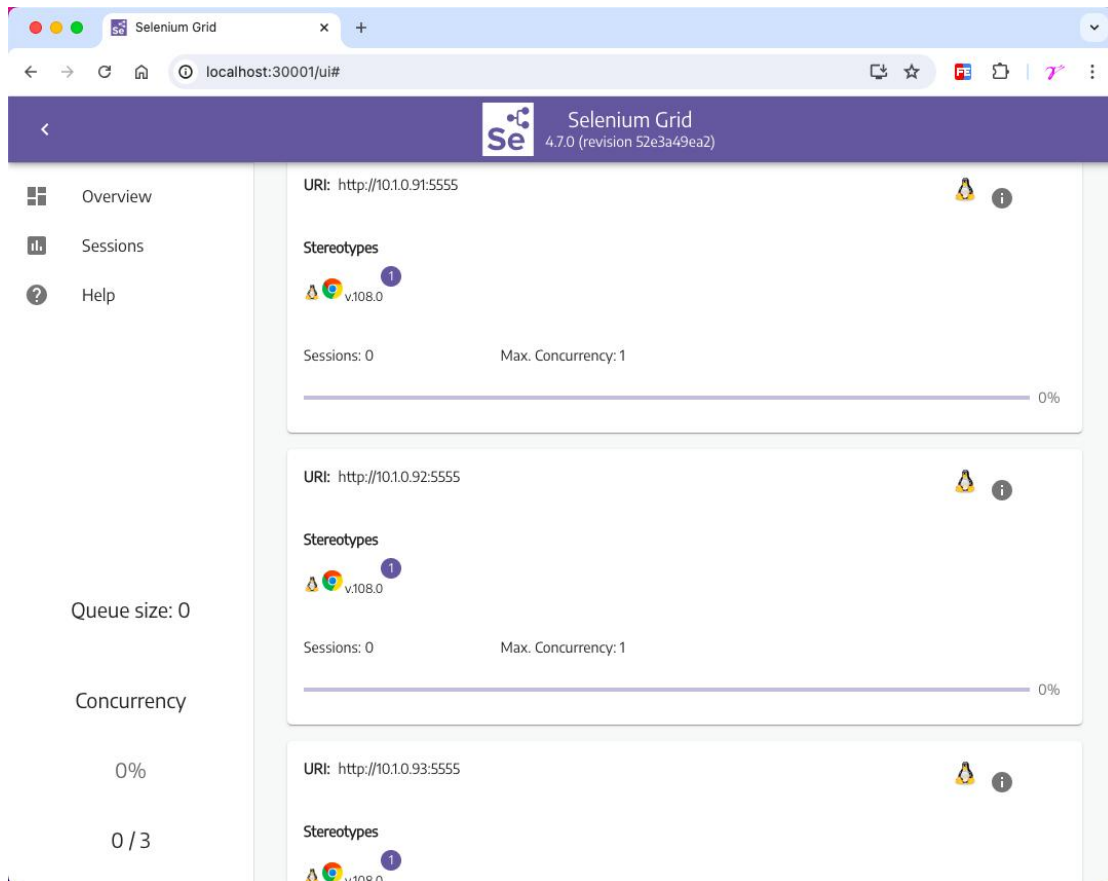


然后通过命令 `kubectl apply -f node-deploy.yaml` 启动 node 容器，如下图：

```
ydjdeMacBook-Air:kind ydj$ kubectl apply -f node-deploy.yaml
deployment.apps/selenium-node-chrome-deploy created
ydjdeMacBook-Air:kind ydj$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
selenium-hub-deploy-87f4d4c47-mmfv  1/1     Running   0           12m
selenium-node-chrome-deploy-77bbb9f854-4w417  1/1     Running   0           6s
selenium-node-chrome-deploy-77bbb9f854-l2c6w  1/1     Running   0           6s
selenium-node-chrome-deploy-77bbb9f854-ssjg8  1/1     Running   0           6s
ydjdeMacBook-Air:kind ydj$
```

启动后使用命令 `kubectl get pods` 查看已经运行的 pods，如上图，可以看到三个 `selenium-node-chrome-deploy-*` 的 pod 的 status 状态是 `Running`，证明刚才的三个 node deploy 创建成功。

再次访问 `grid console` 页面，如下图所示，可以看到注册了三个 node 节点。



总结

Selenium Grid 的从多设备部署、Docker 部署到 Kubernetes 部署的三种进程也是技术发展的一个体现。自身版本也从 1.0、2.0、3.0 发展到现在的 4.0，支持在多个物理机或者虚拟机(跨平台、跨浏览器)上执行大批量的测试用例，从而减少测试执行时间。

Selenium Grid 1.0 是最初的 Selenium Grid 版本，它提供了简单的远程会话并行运行的功能；Grid 2.0 时引入了分布式并行测试的新概念，可以在多个机器上运行测试；Grid 3.0 提供了对 Mobile 和 Desktop 浏览器的并行测试支持，并引入了 Zinc（一种代理服务器）来管理并行测试会话；到 21 年底更新的 4.0，便引入了对 Kubernetes 的支持，使得在容器环境中部署和管理 Selenium Grid 变得更加简单。

Selenium Grid 的版本更迭，都极大地促使着并行测试往更简单、易用的方向发展。部署方式也随着更迭。

本文介绍了三种不同方式的部署，读者可以根据自身使用场景进行选择。但如果你是从事自动化测试工作者，这三种方式就必须掌握，而且更为熟练。

拓展学习

[4] 云端全链路测试技术全栈学习

咨询：微信 atstudy-js 备注：全栈



股市火爆之测试随想

◆ 作者：陆空

今年的“十一”，除了旅游火爆之外，还有另外一件火爆的事件，就是国庆前一周，各个券商 APP 在交易日时无法登录，上证股票交易缓慢，一度处于瘫痪的状态。这个放假前的最后一个周末，上交所进行大交易量的模拟测试。然而，还是无法预计到股民的热情程度，国庆前的最后一个交易日，交易又一度缓慢，再加上国庆期间新开户的股民，上交所在国庆期间，再次启动更大规模的模拟测试。

全民关注的点是股市暴涨、股民的情绪高涨，连国庆放假也在证券所开户开户...而对负责此项目的项目组来说，却是个灭顶性的灾难，此次客户的钱是无法计算的，且此次的问题，不知道存有着多少程序及接口要整改，多少设备资源要调整。

出现的问题：

现象：各个券商的 APP 无法登录，交易缓慢。

原因：同时登录的用户数据量太大，交易操作量过大导致。

根据原因：现在的用户数比 N 年前的已增加了不止几倍，且现在的股票交易执行可量化，当大量的大批量交易执行量化后，交易量就会成几倍，甚至几十倍的增长。

结论：现有的程序逻辑和设备资源无法满足当前的交易市场。

从上面的结论中，可以看出项目组在以下的方面出现了问题：

一、需求方面：随着股票市场需求量的增长，各券商没有重新去评估对应的程序和资源是否符合当前的市场需求，而此次明显地可以看到现有的程序和设备已不适合当前的市场需求。



二、程序方面：程序逻辑中对大数据量用户并发和大批量处理机制处理不完善，导致出现很多账户登录不了，无法交易....

需改进的地方：

一、从需求方面

定期对市场的指标进行调研，根据市场指标，重定义市场的需求，使现有的程序机制匹配当时市场需求，以减少出现市场需求和程序需求严重不匹配而导致故障出现的情况。市场的指标，包括市场用户数、用户活跃数、交易量等主要指标；而市场指标在什么样的变化范围内，才需要进行需求变更，需要各方对指标变化的范围进行定义，从而重新定义需求。

二、从程序方面

第一，程序的设计逻辑需要考虑并发量大时，程序的处理逻辑，着重考虑用户的使用感受。

第二，性能需求，需要做更详细的性能需求分析，考虑到更多的异常情况。且在性能报告中，除了要满足正常的性能需求中的指标外，还需要验证程序和资源在什么样的数据量下会成为瓶颈，供项目组评估后续的调优方案。

改进的措施：

需求方面项目组无法掌控，但程序方面，项目组可以一些具体的改进方案。咱们从以下方面进行分析。

一、测试策略方面：

- 重视用户的使用体验。除了正常的功能测试外，还需要重点考虑特别在程序无法登录、缓慢等异常的情况下，用户的体验是一个重要的验证点。

- 重视性能测试。需要根据性能需求，做更详细的性能需求分析，考虑到更多的异常情况，包括资源、压力和稳定性测试的性能需求分析。且在性能方案和性能报告中，除了要满足正常的性能需求中的指标外，还需要验证程序和资源在什么样的数据量下会成为瓶颈，供项目组评估后续的调优方案。



二、风险监控方面：

项目组在设计方案时，应当考虑为系统增加风险预警和风险监控等基础的功能，设置相关的指标来监控系统，当指标超过阈值时，发出预警，这样子才能及时发现并响应问题。

三、技术方面：

以上述看到的问题的根本原因上来说，主要问题在于优化程序。而优化程序其实就是性能优化、接口优化等。下面详细介绍一下性能优化和接口优化。

- 性能优化：



以一个测试数据库的性能测试为例，在结果中分析存储过程方法中性能的差异性。

性能需求：以 Insert 数据的场景为例，验证在各种并发数下，不同的存储过程方法的性能体现。

数据：初始化 24 张表数据，按主键进行分区。初始化的样式如下图的表 1 所示：

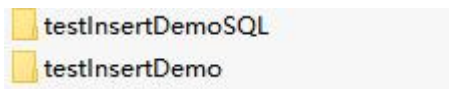
```
drop table table_1 if exists;
CREATE TABLE table_1(
  id varchar(32) NOT NULL,
  key_a integer DEFAULT NULL,
  key_b integer DEFAULT NULL,
  key_c integer DEFAULT NULL,
  key_d integer DEFAULT NULL,
  key_e integer DEFAULT NULL,
  key_f integer DEFAULT NULL,
```



场景：测试存储过程，SQL 两种方式

机器：独立的压测机

脚本：执行这两种方式的测试脚本



压测结果：

存储过程-测试记录

并发数	CPU	CPU (db)	内存 (GB)	RAM (%)	99th (ms)	TPS
	(client)					
10	6.56%	16%	2.8	18.19%	0.8	20600
50	20.12%	38%	5.9	38.53%	1.8	61500
100	30.00%	45%	8.8	57.15%	3.4	85000
200	54.61%	55%	7.9	51.00%	4.1	150000
500	75.75%	57%	11.1	71.00%	8.5	160000
1000	85.96%	60%	10.3	66.03%	14	160000

SQL-测试记录

并发数	CPU	CPU (db)	内存 (GB)	RAM (%)	99th (ms)	TPS
	(client)					
10	5.00%	21%	1.9	12.43%	4.3	12000
50	7.75%	33%	2.4	15.83%	11	19000
100	9.05%	35%	3.8	25.12%	16	21000
200	9.78%	35%	2.6	17.51%	33	21000
500	12.08%	35%	2.8	18.33%	80	20000
1000	12.71%	35%	2.8	18.38%	170	20000

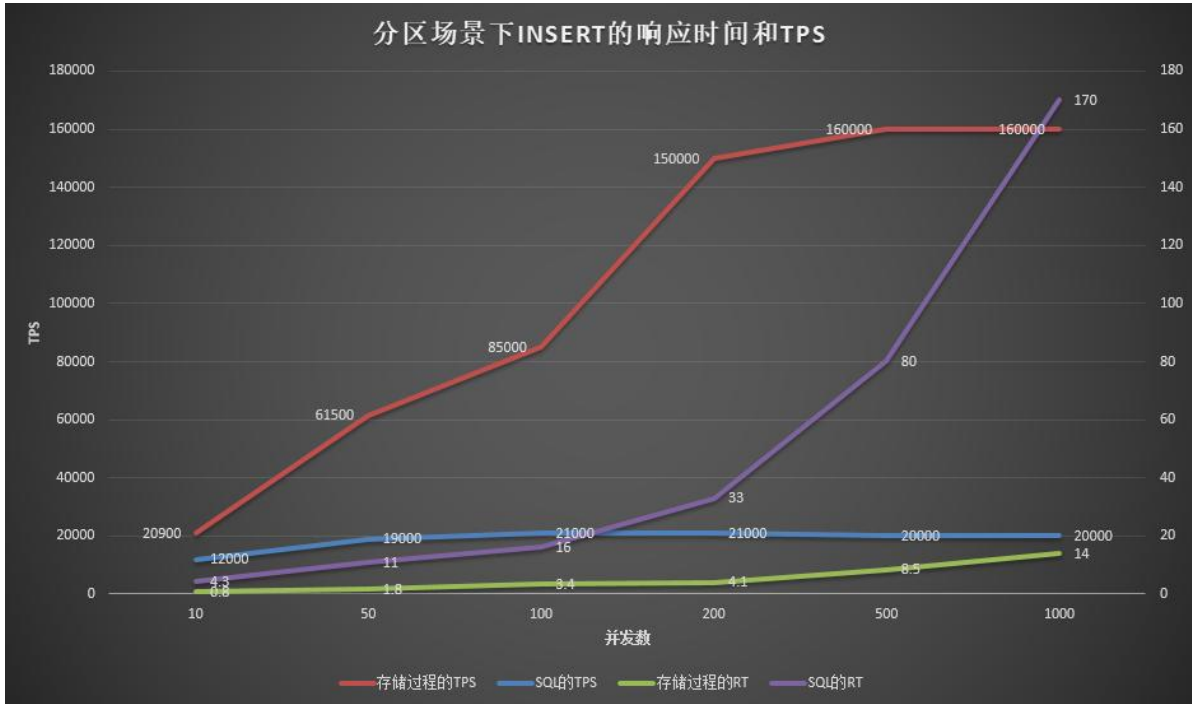
压测结果分析：

1、通过存储过程 INSERT 数据，1000 并发时，压力机的 CPU 使用率已到达 85%，成为当前场景的瓶颈。DB 的 CPU 使用率也到达 60%~70%，系统资源得到充分的利用，未达到当前场景的瓶颈。



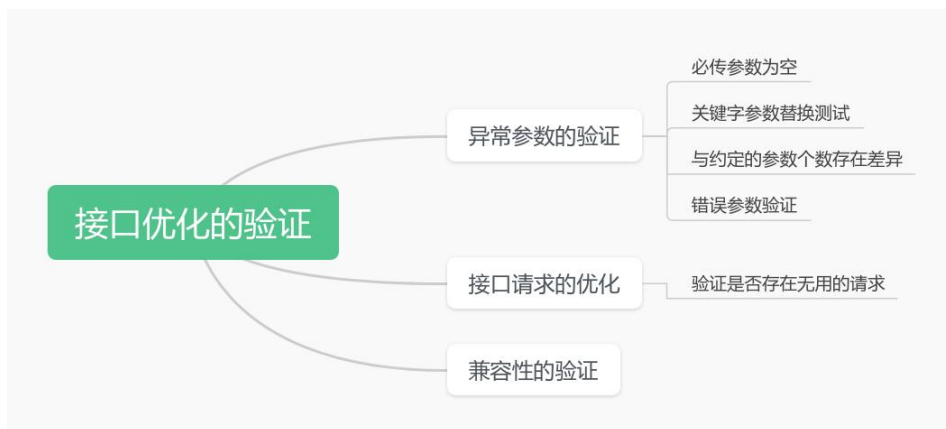
2、通过 SQL INSERT 数据，1000 并发时，压力机和 DB 的 CPU 使用率均未达到瓶颈，并发数的增加，响应时间同步增加，未能有效利用系统资源。

从下图可以看出，相同配置下，使用存储过程进行 INSERT 数据的 TPS 基本稳定在 160000，使用 SQL 进行 INSERT 数据的 TPS 基本稳定在 20000。存储过程的写入性能远远优于通过 SQL 进行写入。

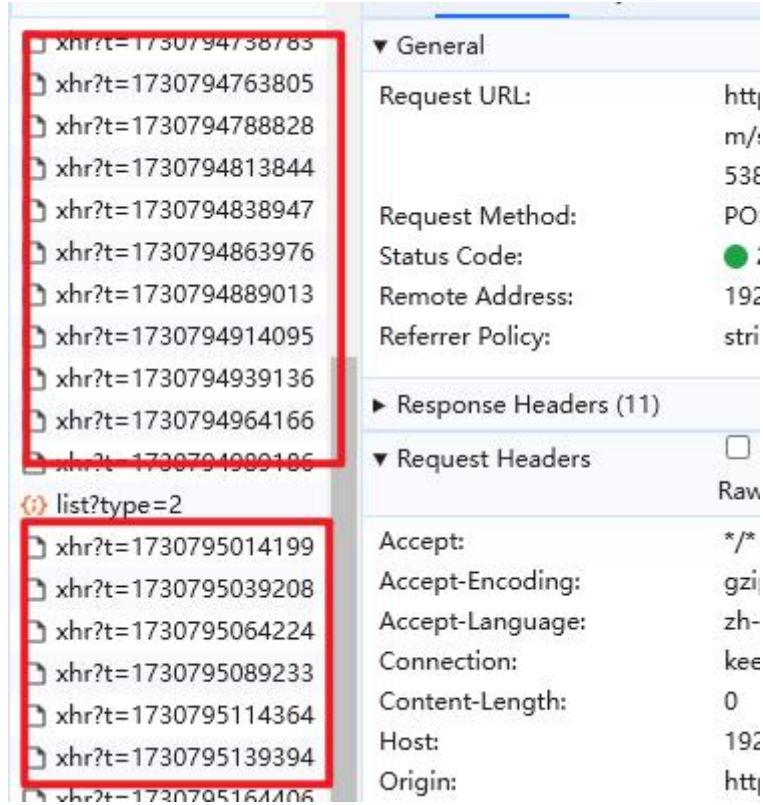


从以上的案例，可以看出相同的条件下，不同的存储过程对同一个数据库性能的影响，因此，在性能调优方面，软件程序的调优的作用和成本远高于硬件系统的调优。

• 接口优化：



接口优化中，除了异常参数的验证和兼容性的验证，这种比较常规的验证点外，接口是否存在重复请求是一个容易被忽视的点。一般只验接口请求是否正确，而像以下这种，一直在发送请求，那需要验证这种请求是否有必要，如有必要，是否有替代的方式，这种请求在无意间占用缓存，从而占用内存。因此，这种接口的优化是非常有必要的。



综上所述，这是我对股市的火爆行情引发的一点关于测试思考，希望能给小伙伴们起到抛砖引玉的作用。



Apifox 接口测试详细使用教程

◆作者：传说三哥

1、Apifox 工具简介

Apifox 接口测试工具是集 API 文档、API 调试、API Mock、API 接口自动化测试等多项实用功能为一体 API 管理平台，定位为?postman+Swagger+mock。通过一套系统、一份数据，解决多个工具之间的数据同步问题。

只需在 Apifox 中定义 API 文档，那么 API 调试、API 数据 Mock、API 自动化测试等功能就可以直接使用，无需再次定义。API 文档和 API 开发调试流程在同一个工具内闭环，API 调试完成后即可确保与 API 文档定义完全一致。高效、及时、准确！

2、Apifox 接口测试工具功能概述



3、Apifox 如何解决接口管理痛点

A.开发人员接口管理使用(Swagger 工具管理接口)

B.后端开发人员通过 postman 工具，一边开发一边测试



C.前端开发人员需要 Mock 工具提供前端调用

D.测试人员通过(postman、jmeter)等工具进行接口测试

为了后台开发、前端开发、测试工程师等不同角色更加便捷管理接口，需要通过一个工具作为载体，Apifox 工具应运而生实现了 API 设计开发测试一体化协作。俗称:Apifox=Postman+Swagger+Jmeter+Mock 工具集成。

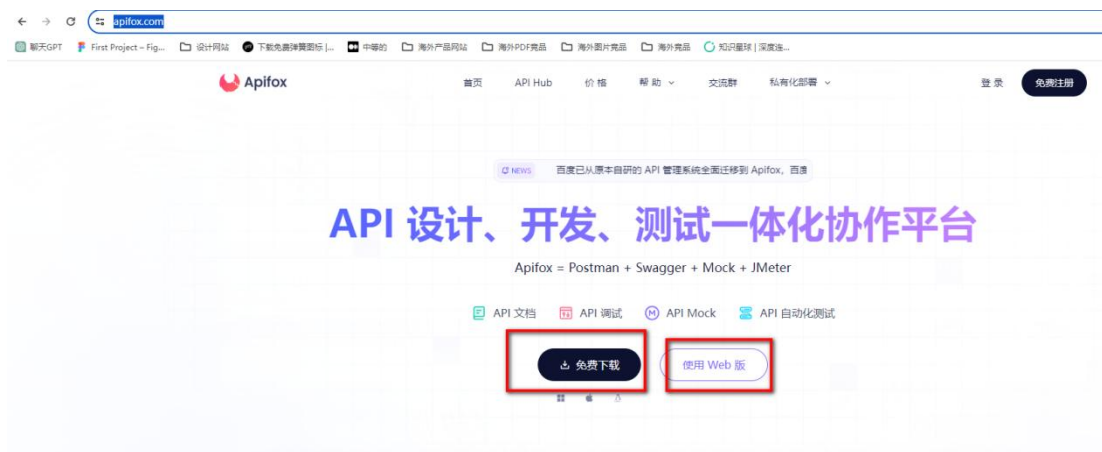
具体分工协助:后端开发人员 Apifox 定义好接口,前端开发人员根据 Apifox 去 Mock 数据,后端开发人员直接在 Apifox 进行接口联调,测试人员直接在 Apifox 工具中编写测试用例,进行接口测试。

优势:通过一套系统、一份数据,解决多个系统之间的数据同步问题。只要定义好接口文档,接口调试、数据 Mock、接口测试就可以直接使用,无需再次定义;接口文档和接口开发调试使用同一个工具,接口调试完成后即可保证和接口文档定义完全一致。高效、及时、准确!

4、Apifox 最新版下载和安装

1.访问 <https://apifox.com/> 官网下载 Apifox 安装包

2.Apifox 工具分为在线网页版和客户端两个版本

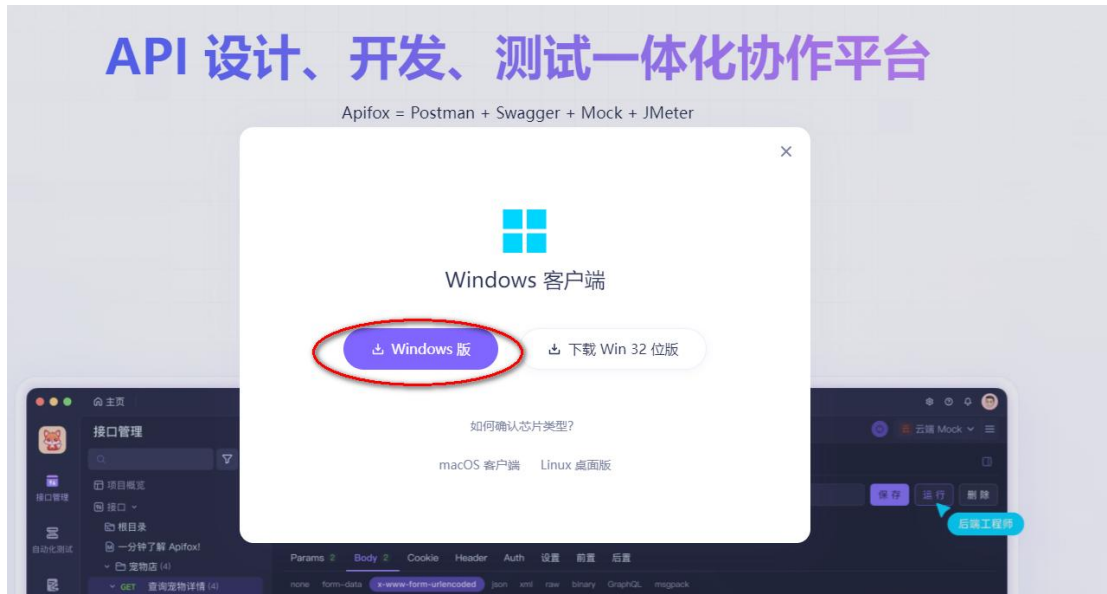


3.Apifox 支持操作系统如下:

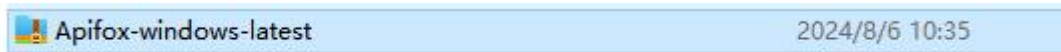
1	Windows 64
2	Windows 32
3	MacOS
4	Linux



4.根据自己操作系统选择对应的版本进行下载(本人电脑 win 10 64 位)



5.下载完成之后，安装包为 zip 格式文件，直接解压

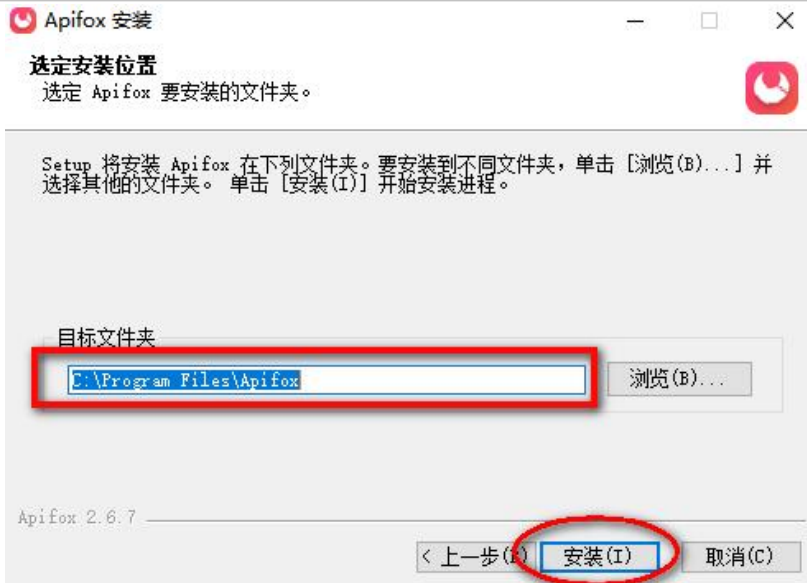


6.解压安装包文件



7.Apifox 具体安装步骤如下:

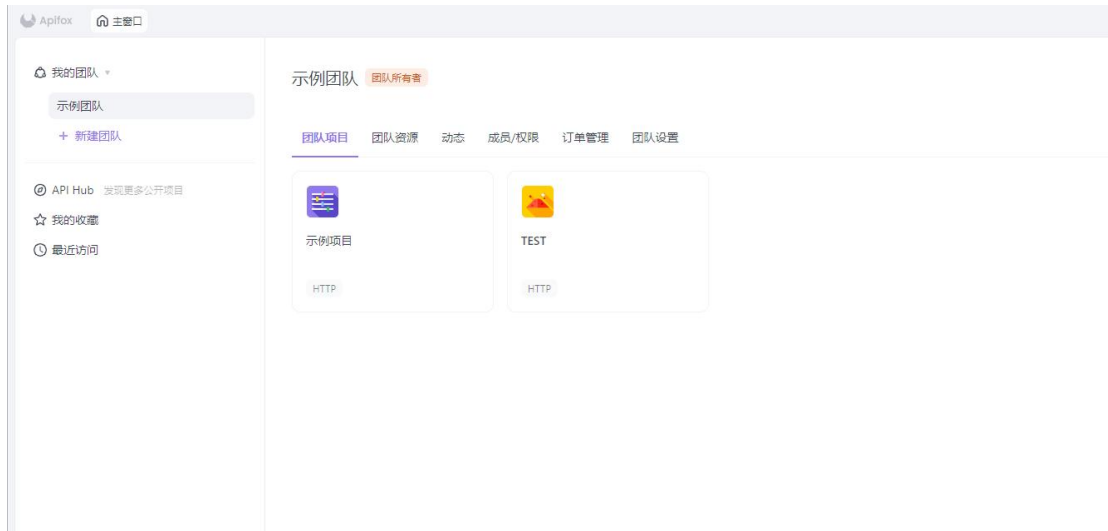




8. 安装成功跳转到登录界面

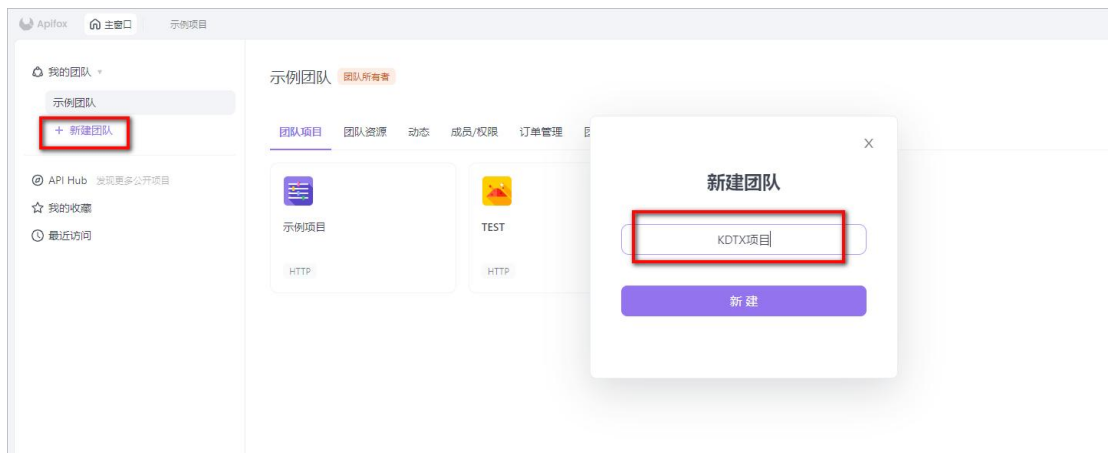


9.微信扫码/手机号/邮箱三种登录方式选择一种方式进行登录。

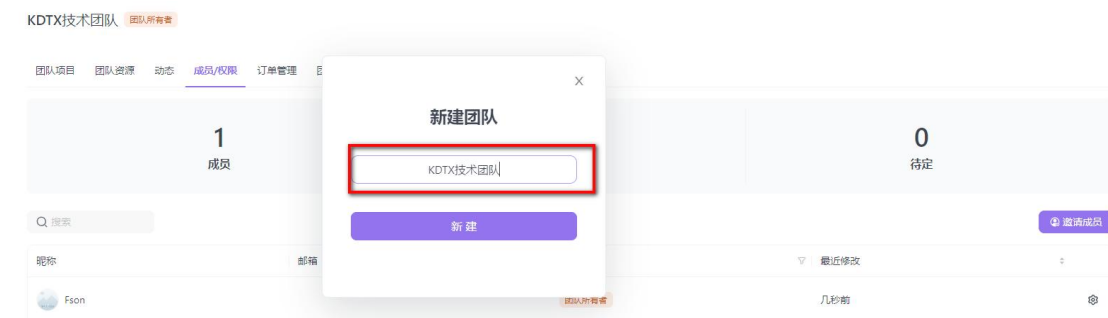


5、Apifox 新建项目流程

1. 点击新建团队，创建新项目



2. 输入自己团队名称，保存



3. 项目概述



6、Apifox 接口测试工具支持协议

接口测试工具对可支持的协议要求越来越高，需要一个工具同时兼容支持多种协议，满足不同的应用场景，apifox 支持的协议远超 postman+jmeter 两种工具协议。

1	HTTP
2	HTTPS
3	TCP
4	RPC
5	Socket
6	GraphQL
7	Dubbo
8	gRPC
9	WebSocket

7、Apifox 接口测试工具支持请求方式

Apifox 工具支持比较常用的 get、post、put 等主流的请求方式外，还支持 delete、options、head、patch、trace、connect、copy 等请求方式。

8、自动生成代码

1. 根据接口模型定义，自动生成各种语言/框架（如 TypeScript、Java、Go、Swift、ObjectiveC、Kotlin、Dart、C++、C#、Rust 等）的业务代码。
2. （如 Model、Controller、单元测试代码等）和接口请求代码。目前 Apifox 支持 130



种语言及框架的代码自动生成。

3. 更重要的是：你可以通过自定义代码模块来生成符合自己团队的架构规范的代码，满足各种个性化的需求。

4. 自定义脚本支持运行 javascript、java、python、php、js、BeanShell、go、shell、ruby、lua 等各种语言代码。

9、兼容多种数据格式

1.支持导出 OpenApi (Swagger)、Markdown、Html?等数据格式，因为可以导出 OpenApi 格式数据，所以你可以利用 OpenApi (Swagger) 丰富的生态工具完成各种接口相关的事情。

2.支持导入 OpenApi (Swagger)、Postman、apiDoc、HAR、RAML、RAP2、YApi、Eolinker、NEI、DOClever、ApiPost、Apizza、ShowDoc、Google Discovery 等数据格式，方便旧项目迁移。

3.支持定时自动导入 OpenApi (Swagger)、apiDoc、Apifox 格式数据。

Apifox 接口测试项目实战部分

下载和安装完 Apifox 工具之后，通过工具 Apifox 进行项目实战对项目接口文档、接口开发、接口调试进行系统讲解，Apifox 开发测试一体化协作完成项目。

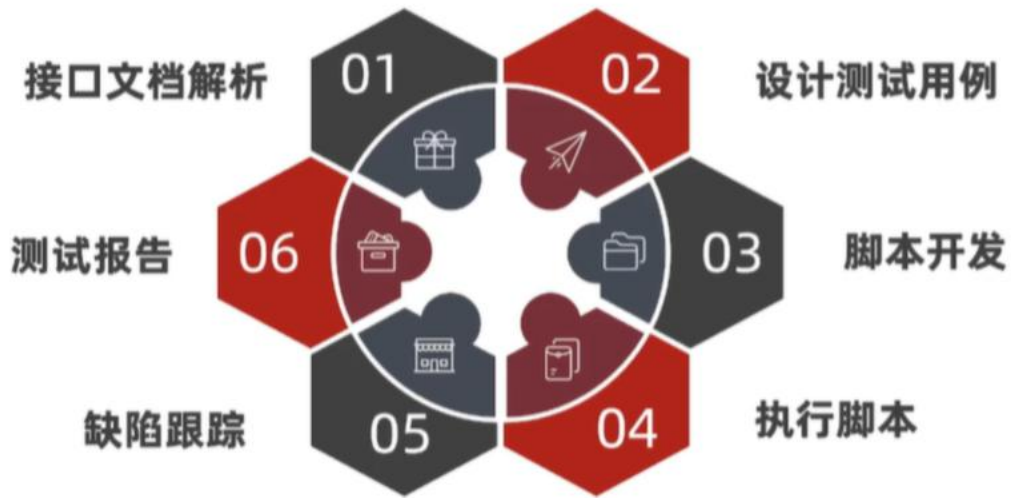
1、接口测试流程

测试流程：

- 开发人员提供接口文档，功能模块、接口请求地址、请求参数、请求方式、响应报文、注意事项
- 设计接口的测试用例
- 测试接口脚本 脚本调试
- 执行脚本
- 缺陷跟踪



- 输出测试报告

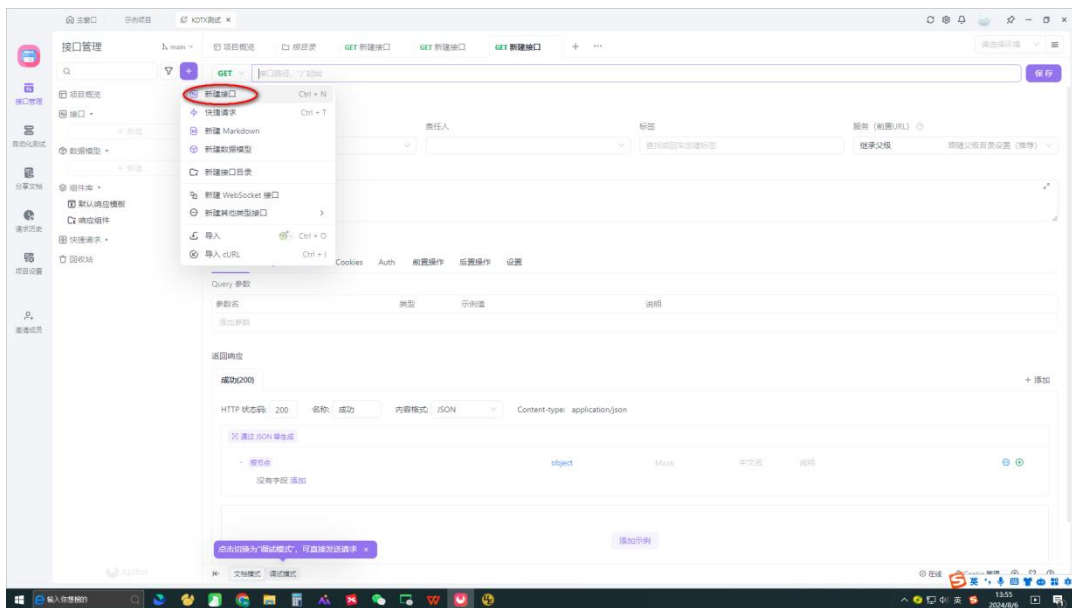


2、Apifox 工具获取登录 uuid 接口开发和调试实战

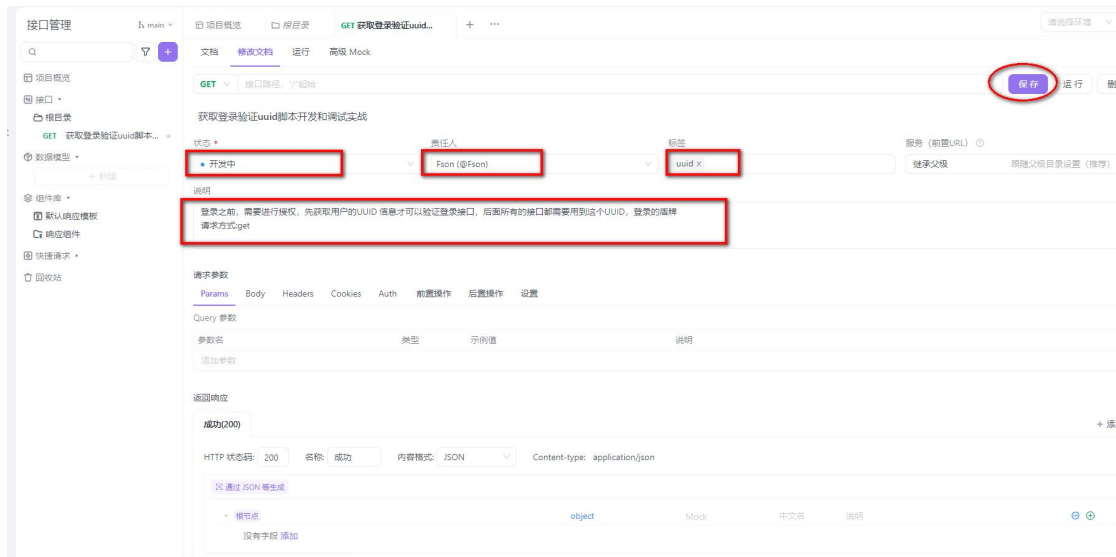
登录之前，需要进行授权，先获取用户的 UUID 信息才可以验证登录接口，后面所有的接口都需要用到这个 UUID，登录的盾牌。

Apifox 工具如何新增新建文档以及编写接口文档信息

1. 点击新建接口，进行编写接口文档



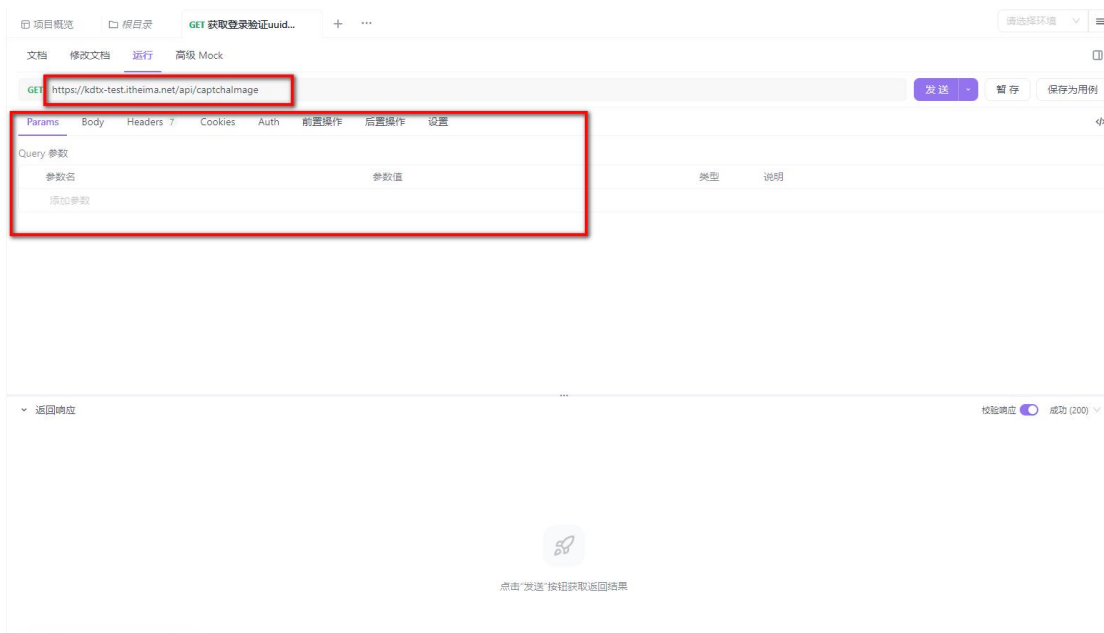
2. 输入接口文档名称、选择【状态】、【责任人】、【输入标签】、【接口说明】等信息，基本信息操作误区之后操作<保存>，登录验证 uuid 接口文档编写完成。



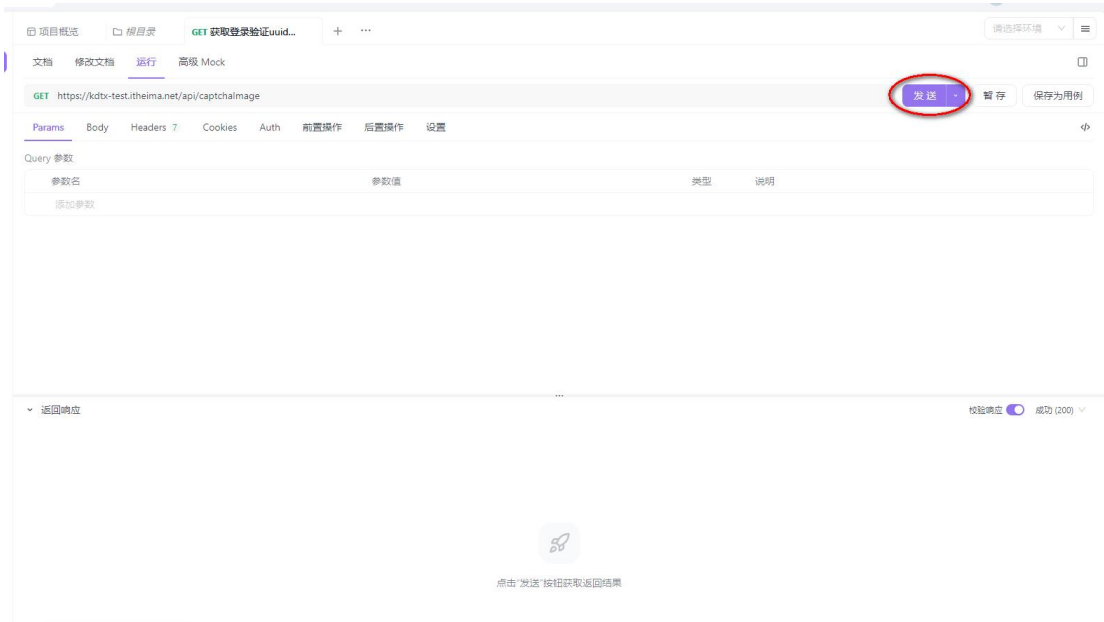
3. 配置获取用户登录 uuid 接口信息

URL	https://kdtx-test.itheima.net/api/captchaImage
请求方式	get

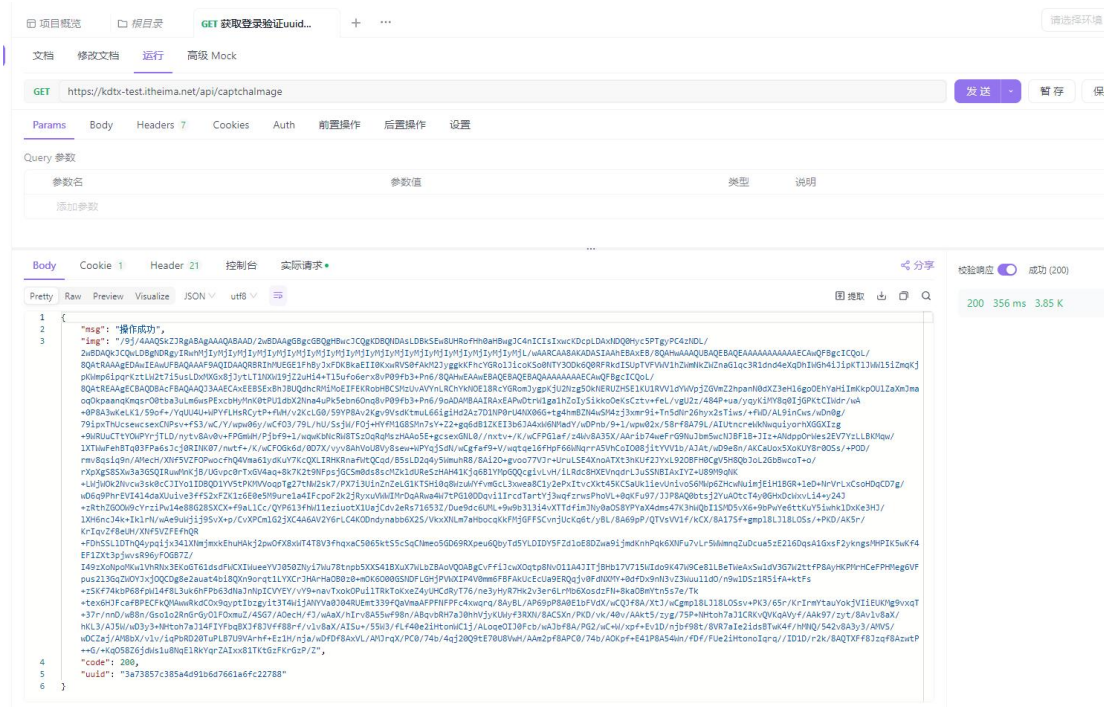
4. 选择<运行>栏目配置，配置获取 UUID 登录接口相关配置信息



5. 配置完成操作<发送>



6. 接口发送之后，服务端响应报文



	2auat4bi8QXn9orqt1LYXCrJHArHaOB0z0+mOK6000GSNDFLGHjPVWxIP4V0mm6FBFAkUcEcUa9ERQqjv0FdNXMY+0dfDx9nN3vZ3Wuu11d0/n9w1DSz1R5ifA+ktFs+zSKf74kbP68fpw14f8L3uk6hFPb63dNaJnNpICVYey/vY9+navTxokOPu1lTRkToKxeZ4yUHCdRyT76/ne3yHyR7Hk2v3er6LrMb6XosdzFN+8ka0BmYtn5s7e/Tk+tex6HJFcafBPECFkQMAwwRkdCOx9qyptIbzgyit3T4WijANYVa0J04RUEmt339fQaVmaAFPFNFpFc4xwqrq/8AyBL/AP69pP8A0E1bFVdX/wCQJf8A/XtJ/wCgmp18LJl8L0Ssv+PK3/65r/KrIrmYtauYokjVIiEUKMg9vxqT+37r/nnD/wB8n/Gso1o2RnGrGy01FOxmuZ/4SG7/AOecH/fJ/wAaX/hIrv8A55wf98n/ABqvbRH7aJ0hhVjyKUWyf3RXN/8ACsXn/PKD/vk/40v/AAkt5/zyg/75P+NHtoh7aJ1CRkVQVKqAVyf/AAk97/zyt/8Av1v8aX/hKL3/AJ5W/wD3y3+NHToh7aJ14FIYfbqBXJf8JVff88rf/v1v8aX/AISu+/55W3/fL40e2iHtonWC1j/ALoqe0IJ0Fcb/wAJbf8A/PG2/wC+W/xpf+Ev1D/njbf98t/8VR7aIe2idsBTwK4f/hMNQ/542v8A3y3/AMVS/wDCZaj/AM8bX/v1v/iqPbRD20TuPLB7U9VArhf+Ez1H/nja/wDfDf8AxVL/AMJrqX/PC0/74b/4qj20Q9tE70U8VwH/AAm2pf8APC0/74b/AOKpf+E41P8A54Wn/fDf/FUE2iHtonoIqrq//ID1D/r2k/8AQTxf8Jzqf8AzwtP++G/+Kq058Z6jdWs1u8NqElRkYqrZAIxx81TKtGzFKrGzP/Z
code	200
uuid	3a73857c385a4d91b6d7661a6fc22788

3、后置操作提取变量

1. 后置操作提取变量作用

在使用 Apifox 进行接口测试的过程中,有时需要将当前测试接口的请求参数提取出来,以使用作其它用途。例如,我们在进行登录接口的测试时,可以通过后置操作获取其相应的 UUID 值,其操作和上文一样,编写相应的接口文档。

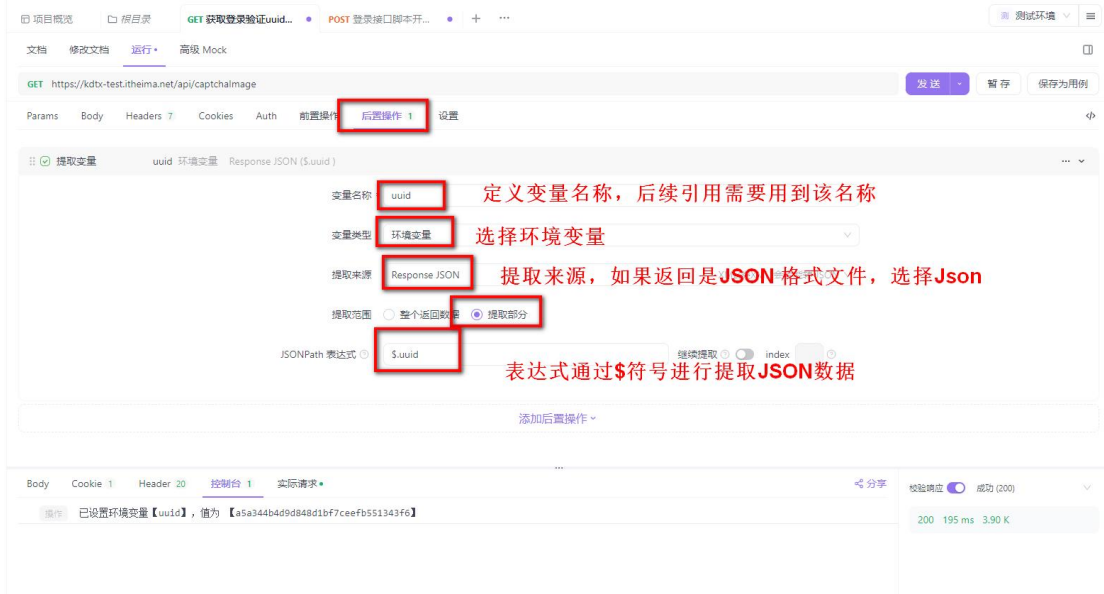
在一些具有明显上下游关系的接口中,有时需要将 A 接口的返回数据作为 B 接口的请求参数。

比如在创建登录信息场景下,需要将获取用户登录 UUID 接口返回的 UUID 中的数据作为后续接口的请求参数,然后在获取 uuid 接口中的「后置操作」中添加「提取变量」功能,基于获取 uuid 接口返回的结果自动提取数据并设置为变量(临时变量/环境变量/全局变量),方便其它接口运行的时候直接使用。

2. 指定变量类型

打开 Apifox 中的某条接口,在“后置操作”页中添加“提取变量”,变量类型选择为“环境变量”。





3. 接口间相互传递数据

例如当前 B 接口的请求参数依赖于 A 接口返回的数据，现希望 B 接口在发送请求的同时能自动获取 A 接口返回的数据并作为请求参数。实现思路如下：

- 1) A 接口在后置操作中添加提取变量功能，将返回的数据自动提取至变量中。
- 2) B 接口对应的参数值直接引用已提取的变量。

2、如何获取 UUID 接口

打开 UUID 接口用例的”后置操作“页，在后置操作中添加提取变量功能。将接口返回 Response JSON 数据里的 uuid 值提取到名为?uuid?的变量中。

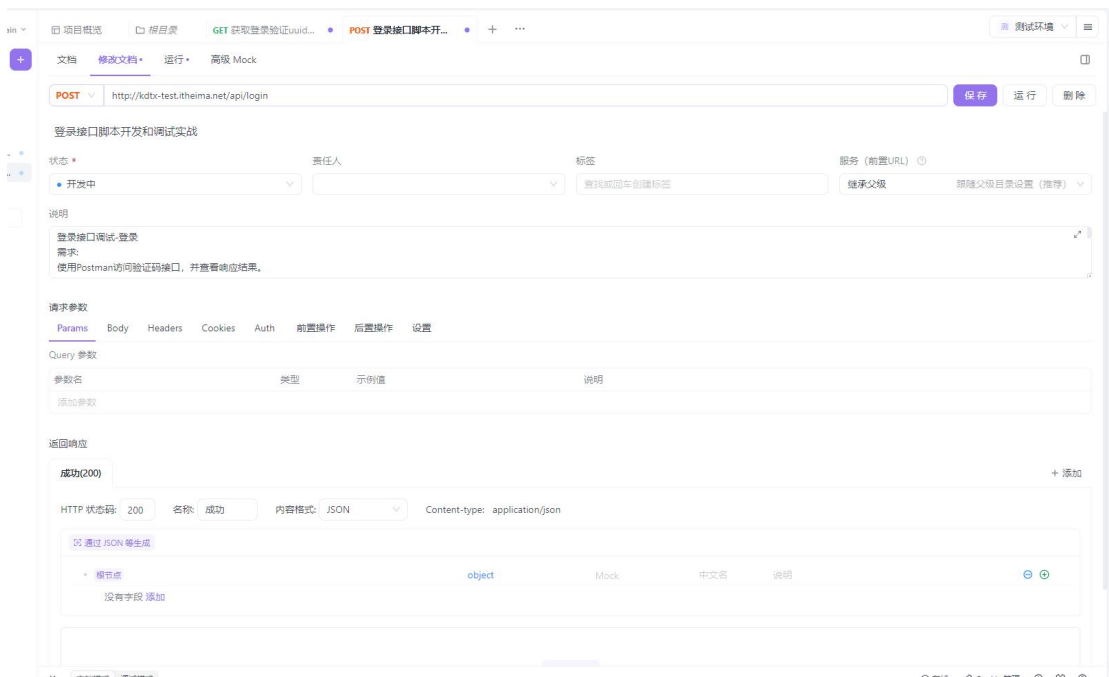
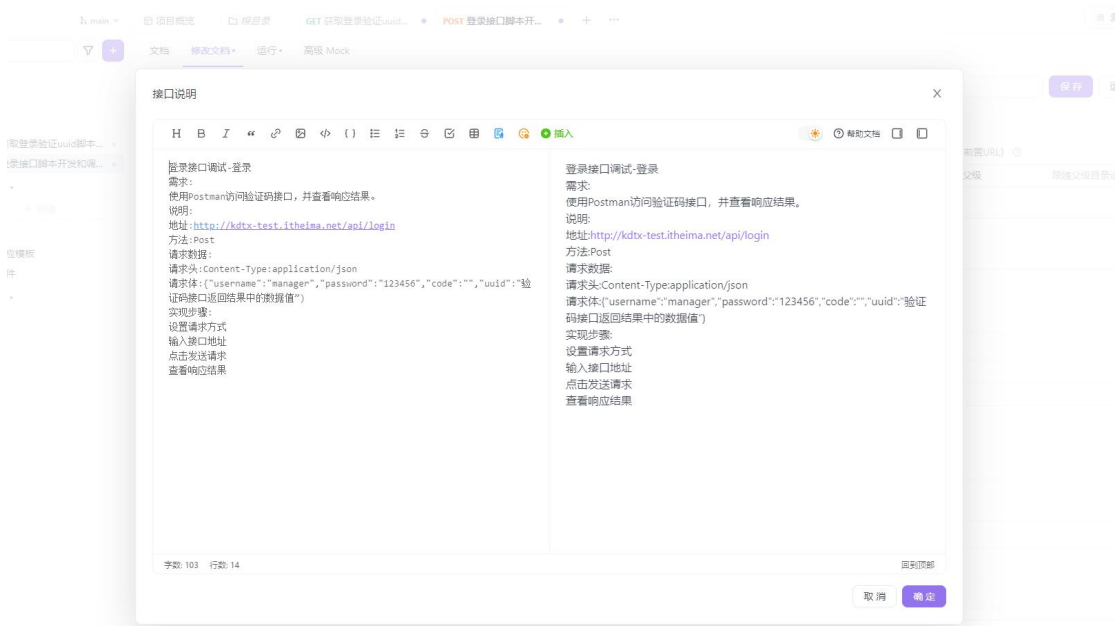
3、登录接口

在登录接口中的请求参数中直接填写 `{{uuid}}`，即可在请求中引用上一步骤中所创建的数值。



4、Apifox 工具管理登录接口脚本开发和调试实战

1. 新建登录接口文档信息



2. 登录接口请求参数

URL	http://kdtx-test.itheima.net/api/login
请求方法	POST
Content-Type	application/json
请求体	<pre>{ "username": "admin", "password": "HM_2023_test", "code": "2", "uuid": "{{uuid}}" }</pre>
备注说明	<p>实现步骤: 设置请求方式 输入接口地址 点击发送请求 查看响应结果</p>

3. Apifox 工具实现登录接口参数配置信息

The screenshot shows the Apifox interface for configuring a login API endpoint. The URL is `http://kdtx-test.itheima.net/api/login` and the method is `POST`. The request body is a JSON object with the following fields:

```
{
  "username": "admin",
  "password": "HM_2023_test",
  "code": "2",
  "uuid": "{{uuid}}"
}
```

Red boxes highlight the URL and the request body configuration in the interface. Red text annotations are present: "请求方式，请求地址信息配置" (Request method, request address information configuration) and "配置登录接口消息体信息" (Configure login interface message body information).



4. 通过后置操作获取 UUID 信息



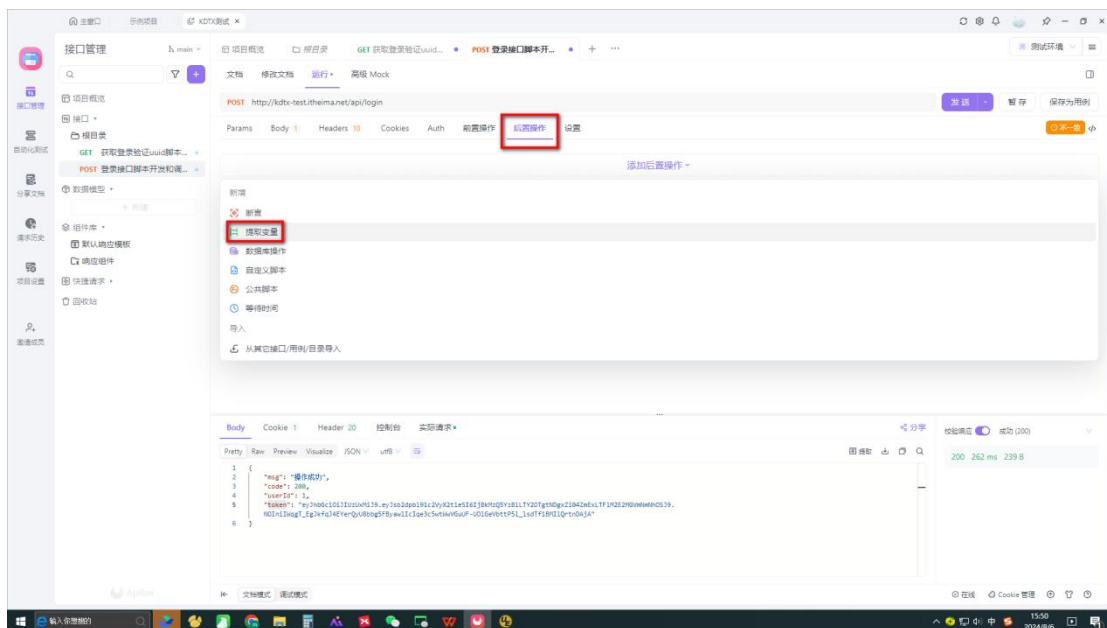
5. 登录接口后置提取 token 信息

由于后续接口都需要依赖登录接口返回的 token 信息，需要将登录接口的返回 token 数据作为后续接口的请求参数。实现思路如下：

1) 登录接口在后置操作中添加提取变量功能，将返回的数据自动提取至变量 login_token 中。

2) 后续接口对应的参数值直接引用 {{login_token}} 已提取的变量。

登录接口运行--->后置操作--->添加后置操作--->选择提取变量

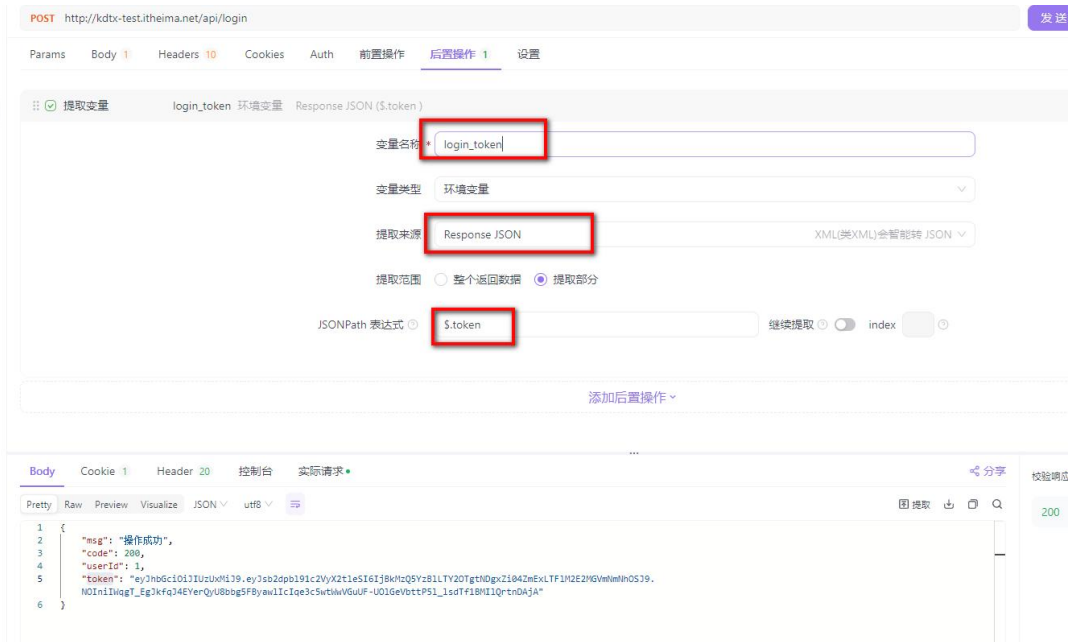


定义变量名称:login_token

变量类型:环境变量

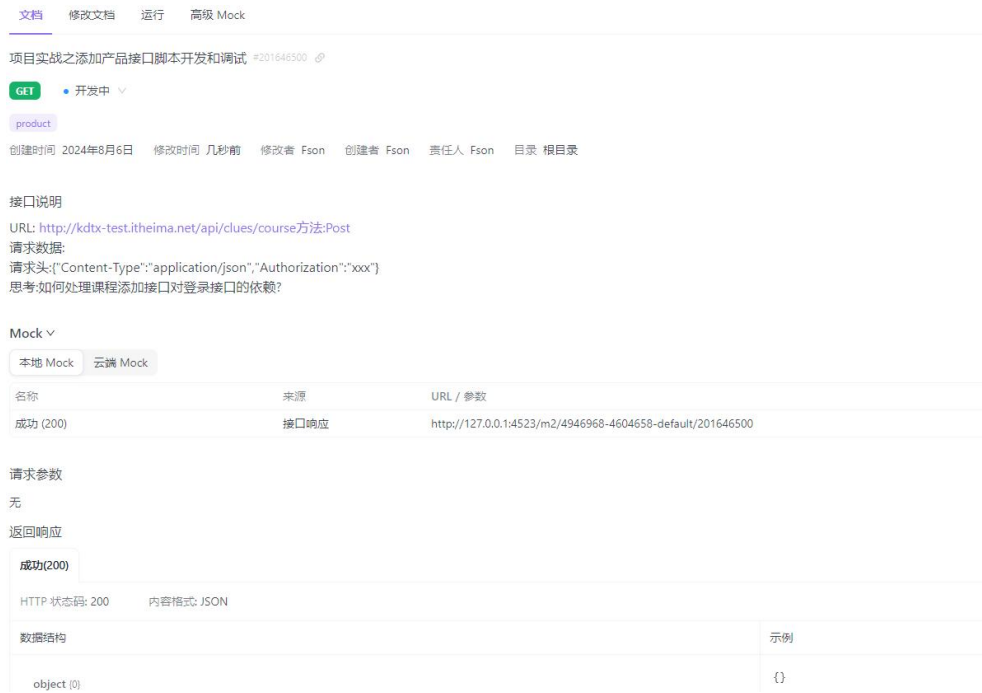
提取来源:Response JSON

JSONPath 表达式:\$.token



5、Apifox 项目实战之添加产品接口脚本开发和调试

1. 新建产品接口文档



2. 添加产品接口请求参数

URL	http://kdtx-test.itheima.net/api/clues/course
请求方法	POST
Content-Type	"application/json","Authorization":"xxx"
请求体	{ "name": "测试开发提升课 01", "subject": "6", "price": 899, "applicablePerson": "2", "info": "测试开发提升课 01" }
备注说明	实现步骤: 设置请求方式 输入接口地址 点击发送请求 查看响应结果 Authorization=token

3. Apifox 工具实现添加产品接口参数配置信息

请求 URL:http://kdtx-test.itheima.net/api/clues/course 请求方式:post

请求头信息配置 Content-Type:application/json,Authorization 字段信息为登录接口返回的 token 字段信息,引用{{login_token}}, 具体配置如下:



4. 添加产品接口请求信息配置如下:

```
{
  "name": "测试开发提升课 01",
  "subject": "6",
  "price": 899,
  "applicablePerson": "2",
```



```
"info": "测试开发提升课 01"
}

POST http://kdtx-test.itheima.net/api/clues/course

Params Body 1 Headers 11 Cookies Auth 前置操作 后置操作 设置

none form-data x-www-form-urlencoded json xml raw binary GraphQL msgpack

动态值
1 {
2   "name": "测试开发提升课01",
3   "subject": "6",
4   "price": 899,
5   "applicablePerson": "2",
6   "info": "测试开发提升课01"
7 }
```

5. 发送请求返回响应结果报文信息

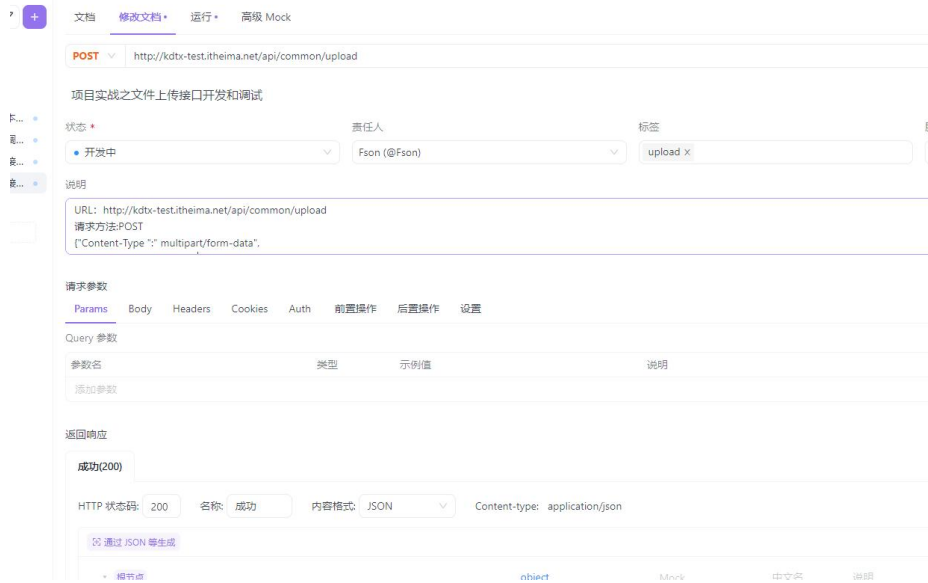
```
{
  "msg": "操作成功",
  "code": 200,
  "data": {
    "id": 23056458704402138
  }
}
```

7、Apifox 项目实战之文件上传接口脚本开发和调试

接口测试过程中需要上传文件操作，通过文件上传接口讲解 apifox 工具是怎么对文件上传进行操作。

1. 新建文件上传接口文档





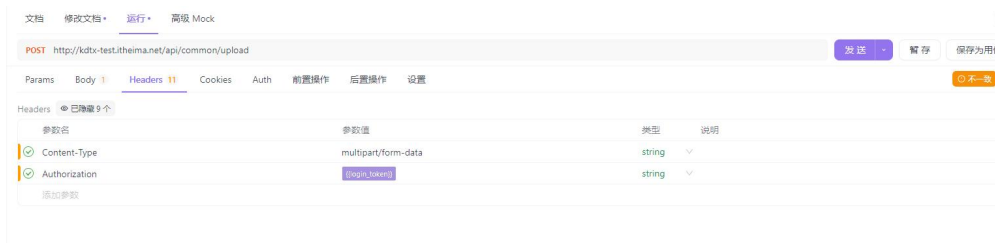
2. 文件上传接口请求参数

URL	http://kdtx-test.itheima.net/api/common/upload
请求方法	POST
Content-Type	{"Content-Type ":" multipart/form-data", "Authorization":"{{token}}"} }
请求体	file
备注说明	实现步骤: 设置请求方式 输入接口地址 点击发送请求 查看响应结果 Authorization=token

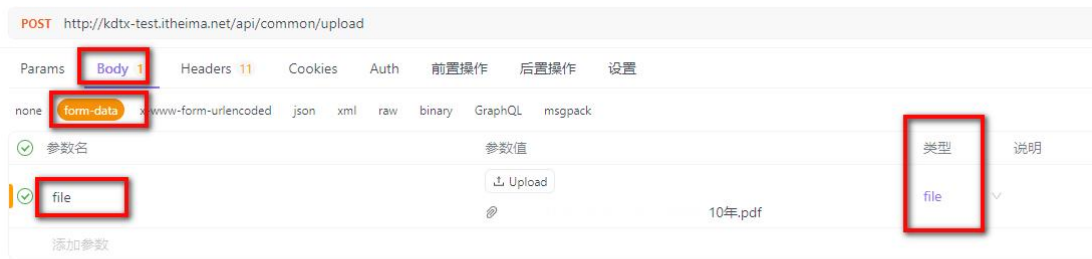
3. Apifox 工具实现上传文件接口参数配置信息

请求 URL:http://kdtx-test.itheima.net/api/common/upload 请求方式:post

请求头信息配置 Content-Type:multipart/form-data,Authorization 字段信息为登录接口返回的 token 字段信息,引用{{login_token}}, 具体配置如下:



4. 上传文件接口请求信息配置，选择 **Body**，选中 **form-data**，参数名称=**file**，参数值=选择需要上传的文件，类型=**file**



5. 发送请求返回响应结果报文信息

```
{
  "msg": "操作成功",
  "fileName": "/profile/upload/2024/08/06/74962262-fdf6-48b0-a59d-a38fa99b1531.pdf",
  "code": 200,
  "url":
"http://localhost:8080/profile/upload/2024/08/06/74962262-fdf6-48b0-a59d-a38fa99b1531.pdf"
}
```

8、总结

总的来说，**Apifox** 这个工具的功能性还是很强大的，很多企业也开始使用这个工具，对于后台开发、前端开发、测试工程师来说，不善于接触新事物对自身的发展是不利的，这篇文章主要介绍了这个工具的基本使用。

拓展学习

[5] **【Python 自动化测试学习交流群】** 学习交流

咨询：微信 **atstudy-js** 备注：学习群



Oracle 数据库分页查询的优化

◆ 作者： fhh192

一、Oracle 数据库的分页操作

对于 Oracle 数据库的分页查询语句一般采用如下的方式。

```
select col1,col2,...,coln
from
(
  select col1,col2,...,coln
  from
  (
    select t.col1,t.col2,...,t.coln,rownum rnum
    from
    (
      --此处为需要进行分页的SQL语句
    ) t
  )
  where rownum<=${分页结束}
)
where rnum>=${分页起始};
```

二、Oracle 数据库的分页操作的性能分析和调优

按照分页操作的 SQL 语句的复杂度，可以将分页操作的种类分为只包含排序操作的分页、包含过滤操作和排序操作的单表查询分页、包含过滤操作和排序操作的单分区表查询分页、包含过滤操作和排序操作的表关联查询分页等几种类型。

1.只包含排序操作的分页

一般地，分页操作的 SQL 语句带有排序操作，即：**order by**。例如：在客户访问记录表中查询最新访问的 30 个客户，这就需要按照客户的访问时间进行降序排序。

对于这类分页操作的调优，需要利用索引是有序的特性，为 **order by** 中的排序列设置



索引。

如果是对多个列进行排序，需要为所有的排序列创建一个复合索引，创建时需要注意复合索引中的各列的顺序需要与 `order by` 中指定的各排序列的顺序一致，且还需注意索引是升序还是降序，以下示例 1 展示了该类 SQL 语句的调优。

【示例 1】

对名为 `user` 的用户信息表执行分页查询，查询最近访问但积分最低的前 10 位用户，分页语句如下所示。

`user` 表中的记录数为 300 万左右。

```
select username,userscore,lasttime
from(
  select username,userscore,lasttime
  from(
    select t.username,t.userscore,t.lasttime,rownum rnum
    from(
      select username,userscore,lasttime
      from user
      order by userscore,lasttime desc
    ) t
  )
  where rownum<=10
)
where rnum>=1;
```

为排序列创建一个复合索引，`userscore` 列在前，`lasttime` 列在后，且将 `lasttime` 列的索引设置为降序，即：

```
create index idx_us_ut on user(userscore,lasttime desc);
```

创建索引后，该语句的真实执行计划中的核心内容如下图所示。

Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		10
*1	VIEW		1	10	10
*2	COUNT STOPKEY		1		10
3	VIEW		1	100k	10
4	COUNT		1		10
5	VIEW		1	100k	10
6	TABLE ACCESS BY INDEX ROWID	USER	1	100k	10
7	INDEX FULL SCAN	IDX_US_UT	1	100k	10

Predicate Information (identified by operation id):

- 1 - filter("RNUM">=1)
- 2 - filter(ROWNUM<=10)



通过以上的执行计划可以知：

(1) 该 SQL 语句执行时访问 User 表时采用了 INDEX FULL SCAN (索引快速全扫) 的访问方式 (因为未指定过滤条件, 所以只能采用 INDEX FULL SCAN 的访问方式);

(2) 执行 INDEX FULL SCAN 时访问 10 条记录后, 已经满足分页要求, 扫描终止, 该 SQL 语句执行完毕。即上图执行计划中的 COUNT STOPKEY (COUNT STOPKEY 一旦获取到满足分页条件所需的记录后即停止 SQL 语句的执行) 操作;

(3) 执行计划中没有诸如 SORT ORDER BY、SORT ORDER BY ROWNUM 之类的排序操作, 说明以上 SQL 语句执行时利用了索引消除了排序操作。

综上所述可知, 该 SQL 的执行计划是正确的, 高效的。

此外, 对于只有 1 个列的降序排序操作, 也可以不将该列的索引设置为降序索引, 而是创建一般的升序索引, 在查询执行时使用 index_desc 提示器, 使得执行对该索引的扫描时采用降序扫描的方式。

如果创建符合索引时, 排序列的顺序与 order by 中的排序列顺序不一致, 在查询执行时, 排序列的索引将仅用于通过索引访问记录, 之后对获取的结果集再执行一次额外的排序操作, 执行计划中将出现 SORT ORDER BY 或 SORT ORDER BY ROWNUM 之类的排序操作提示, 说明并未利用索引消除排序操作。

如果创建的索引的顺序与实际排序的顺序不一致, 例如: 降序排序, 升序索引, 执行时也未使用 index_desc 提示器, 则在查询执行时, 排序列的索引也将仅用于通过索引访问记录, 之后也要对获取的结果集再执行一次额外的排序操作。

2. 包含过滤条件和排序操作的分页

对于包含过滤条件和排序的分页, 除了为排序列创建索引外, 还需要为过滤条件创建索引。根据过滤条件的不同, 又分为等值过滤和非等值过滤。

I. 等值过滤

如果分页语句中包含等值过滤和排序, 此时需要为等值过滤列和排序列创建复合索引, 该复合索引的顺序为等值过滤列在前, 排序列在后, 即:

(等值过滤列 1, 等值过滤列 2, ..., 排序列 1, 排序列 2, ...)



以下示例 2 展示了该类 SQL 语句的调优。

【示例 2】

对名为 user 的用户信息表执行分页查询，查询最近访问但积分最低的前 10 位类别为 C 类用户，分页语句如下所示。

```
select username,userscore,userclass,lasttime
from(
  select username,userscore,userclass,lasttime
  from(
    select t.username,t.userscore,t.userclass,t.lasttime
    from(
      select username,userscore,userclass,lasttime
      from user
      where userclass='C'
      order by userscore,lasttime desc
    ) t
  )
  where rownum<=10
)
where rnum>=1;
```

为等值过滤列，排序列创建一个复合索引，按 userclass, userscore, lasttime 的顺序创建，即：

```
create index idx_us_ut on user(userclass,userscore,lasttime desc);
```

创建索引后，该语句的真实执行计划中的核心内容如下图所示。

Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		10
*1	VIEW		1	10	10
*2	COUNT STOPKEY		1		10
3	VIEW		1	15981	10
4	COUNT		1		10
5	VIEW		1	15981	10
6	TABLE ACCESS BY INDEX ROWID	USER	1	15981	10
*7	INDEX RANGE SCAN	IIDX_US_UT	1	15981	10

- 1 - filter("RNUM">=1)
- 2 - filter(ROWNUM<=10)
- 7 - access("USERCLASS"='C')



通过以上的执行计划可以知：

(1) 该 SQL 语句执行时访问 User 表时采用了 INDEX RANGE SCAN（索引范围扫描）的访问方式，因为已经指定了过滤条件，且为过滤条件列设置了索引。

(2) 执行 INDEX RANGE SCAN 时访问 10 条记录后，已经满足分页要求，扫描终止，该 SQL 语句执行完毕。即上图执行计划中的 COUNT STOPKEY 操作；

(3) 执行计划中没有诸如 SORT ORDER BY、SORT ORDER BY ROWNUM 之类的排序操作，说明以上 SQL 语句执行时利用了索引消除了排序操作。

综上所述可知，该 SQL 的执行计划是正确的，高效的。

需要注意的是，创建复合索引时等值过滤条件列要放在排序列之前，如果将其放在排序列之后，即：按 userscore, lasttime, userclass, 的顺序创建上述分页语句的真实执行计划将变为如下图所示的。

Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		10
*1	VIEW		1	10	10
*2	COUNT STOPKEY		1		10
3	VIEW		1	15981	10
4	COUNT		1		10
5	VIEW		1	15981	10
6	TABLE ACCESS BY INDEX ROWID	USER	1	15981	10
*7	INDEX FULL SCAN	IDX_US_UT	1	15981	10

```

1 - filter("RNUM">=1)
2 - filter(ROWNUM<=10)
7 - access("USERCLASS"='C')
   - filter("USERCLASS"='C')
    
```

通过以上的执行计划可以发现，对 user 表的访问方式由 INDEX RANGE SCAN 变为 INDEX FULL SCAN 方式。

过滤执行时为执行计划中 id 为 7 中对应的 access 和 filter 两种方式的结合，该方式说明在使用 userclass 列上索引访问 user 表时在执行索引扫描的同时又执行了对记录的过滤操作。具体操作为：扫描索引时，同时对每条索引按照过滤条件，如果符合过滤条件的记录达到分页操作指定的 10 条后，扫描停止。

这种边扫描边过滤的访问方式的执行效率较直接执行 INDEX RANGE SCAN 的访问方式稍低，因为这种方式的逻辑读多于 INDEX RANGE SCAN 方式。



II.等值过滤和非等值过滤

如果分页语句中包含等值过滤非等值过滤和排序，创建复合索引时，需要将等值过滤列放在前部，排序列放在中部，非等值过滤列放在后部，即：

(等值过滤列 1,等值过滤列 2,...,排序列 1,排序列 2,...,非等值过滤列 1,非等值过滤列 2...)

同样地，如果分页语句中只有非等值过滤和排序，则需要将非等值过滤列放到复合索引的后部，以下示例 2 展示了该类 SQL 语句的调优。

【示例 3】

对名为 user 的用户信息表执行分页查询，查询最近访问但积分最低的且注册时间大于 18 个月的前 10 位类别为 C 类用户，分页语句如下所示。

```
select username,userscore,userclass,regtime,lasttime
from(
  select username,userscore,userclass,regtime,lasttime
  from(
    select t.username,t.userscore,t.userclass,t.regtime,t.lasttime,rownum rnum
    from(
      select username,userscore,userclass,lasttime
      from user
      where userclass='C' and regtime>18
      order by userscore,lasttime desc
    ) t
  )
  where rownum<=10
)
where rnum>=1;
```

为等值过滤列，非等值过滤列，排序列创建一个复合索引，按 userclass，userscore，lasttime，regtime 的顺序创建，即：

```
create index idx_us_ut on user(userclass,userscore,lasttime desc,regtime);
```

创建索引后，该语句的真实执行计划中的核心内容如下图所示。



Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		10
*1	VIEW		1	10	10
*2	COUNT STOPKEY		1		10
3	VIEW		1	15981	10
4	COUNT		1		10
5	VIEW		1	15981	10
6	TABLE ACCESS BY INDEX ROWID	USER	1	15981	10
*7	INDEX RANGE SCAN	IIDX_US_UT	1	15981	10

Predicate Information (identified by operation id):

- 1 - filter("RNUM">=1)
- 2 - filter(ROWNUM<=10)
- 7 - access("USERCLASS"='C' AND "REGTIME">18)
- filter("REGTIME">18)

通过以上的执行计划可以知,该分页 SQL 查询执行时已经消除了排序操作,且对 user 的访问方式为 INDEX RANGE SCAN (索引范围扫描),因为创建复合索引时将非等值过滤列 regtime 放到了排序列之后,所以执行 INDEX RANGE SCAN 时又进行了过滤操作,虽然这种方式的执行性能稍低于直接执行 INDEX RANGE SCAN,但如果创建复合索引时将非等值过滤列 regtime 放到了排序列之后,该分页 SQL 查询的真实执行计划将变为如下图所示的形式。

Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		10
*1	VIEW		1	10	10
*2	COUNT STOPKEY		1		10
3	VIEW		1	6158	10
4	COUNT		1		10
5	VIEW		1	6158	10
6	SORT ORDER BY		1	6158	10
7	TABLE ACCESS BY INDEX ROWID	USER	1	6158	6389
*8	INDEX RANGE SCAN	IIDX_US_UT	1	6024	6389

Predicate Information (identified by operation id):

- 1 - filter("RNUM">=1)
- 2 - filter(ROWNUM<=10)
- 8 - access("USERCLASS"='C' AND "REGTIME">18)

该执行计划中出现了 SORT ORDER BY 排序操作,且执行 INDEX RANGE SCAN 时扫描了 6389 条记录,说明查询执行时未利用索引是有序的特性,对查询结果结果集又执行了排序操作并只获取了前 10 条记录,显然,这种方式的执行效率和性能相比 INDEX RANGE SCAN 和过滤方式是较差的。



3.包含过滤操作和排序操作的单分区表查询分页

上述介绍的通过创建适当的索引调优分页操作的方法同样适用于分区表的分页查询。

Oracle 数据库的分区表的索引分为本地索引 (Local Index) 和全局 (Global Index) 索引两种类型。

本地索引 (Local Index) 的分区与表的分区完全相同, 每个分区都有自己的本地索引分区。本地索引的维护由 Oracle 自动进行, 通常本地索引的维护代价较低。

全局索引 (Global Index) 与表的分区无关, 因此它只有一个分区。全局索引的维护和更新对于表的分区变化的响应速度可能较慢, 但是在全局索引分区上可以执行跨分区的查询操作。

通过以上对两种索引的介绍可知, 为分区表创建索引时需要在本地索引与全局索引中做出选择, 选则原则如下。

(1) 如果分页查询操作只访问一个分区, 则可以为相应的列创建本地索引;

(2) 如果分页查询操作需要访问多个分区 (跨分区访问), 则需要为相应的列创建全局索引, 否则无法确保分页操作中排序操作的顺序也索引的顺序一致, 从而无法利用索引的有序性消除分页中的排序操作;

(3) 在分页查询操作中没有过滤条件的情景下, 查询执行时将扫描分区表中的所有分区, 如果该分页查询中的排序列就是创建分区表时的范围分区列, 且范围分区中每个分区中的数据是递增的, 此时可以为排序列创建本地索引。如果创建的分区是 HASH 分区或 LIST 分区, 因为这两类分区是无序的, 所以需要为排序列创建全局索引。

4.表关联查询分页

对于多表关联排序的调优, 需要遵循以下的原则。

(1) 将表关联方式调整为嵌套循环关联, 需要对哪个表排序, 则应该该表作为驱动表, 注意驱动表返回的数据顺序应与排序的顺序一致。如果采用外连接, 则采用嵌套循环关联时, 驱动表只能为主表;

(2) 根据 (1) 可知, 排序列只来自于嵌套循环关联的驱动表, 即: 只能对一个表排序, 如果排序列即来自驱动表也来自被驱动表, 则需要等表关联全部执行完成后, 对



结果集执行一次排序，导致排序操作无法消除；

(3) 关联分页查询语句中不能出现 max、min、avg 之类的聚合函数，也不能出现去重 (distinct)、分组 (group by)、并 (union/union all) 等操作，这些操作会使得全部关联操作完成之后再分页，执行性能较低。

【示例 4】

对 CD_U_MC_MODEL_REMINDER_ORGCUST (A) 和 CD_U_MC_MODEL_REMINDER (B) 表执行关联，并进行分页操作，排序后获取前 50 条记录，SQL 语句如下。

```
select * from(
  select tmp.*,rownum row_id
  from(
    select a.mid,a.cno,a.cname,b.stu
    from CD_U_MC_MODEL_REMINDER_ORGCUST a inner join CD_U_MC_MODEL_REMINDER b
    on a.model_id=b.model_id and b.status='01'
    where a.procod='24' and a.assign_orgno in ('992400000000') and
    (b.role_id_id='BASEPF0015' or b.role_id_id is null) and b.model_id='xnscs1000000008101'
    order by a.reach_date desc
  ) tmp
  where rownum<=50
)
where row_id>=1;
```

其中 CD_U_MC_MODEL_REMINDER_ORGCUST 表记录较多，达到 3 千万左右，该 SQL 语句执行较慢，平均执行用时 16 秒左右。

经对执行计划进行分析，语句中存在 SORT ORDER BY ROWNUM 排序操作，该 SQL 执行表表关联后，返回的结果集较大，达到 300 万条记录，对 300 万条进行排序，造成了较大的开销，且因为 PGA 空间有限，排序占用了临时表空间，同时造成了一定的 I/O 开销（出现 direct path readtemp 和 direct path write temp 事件）。

对该 SQL 查询调优的首要就是消除排序，因排序列为 CD_U_MC_MODEL_REMINDER_ORGCUST 表，所以，为该表创建一个联合索引，按“(等值过滤列,关联列,排序列)”的顺序依次创建，需要注意的是，该 SQL 查询按 CD_U_MC_MODEL_REMINDER_ORGCUST 表的 reach_date 列降序排序，reach_date 列需要创建降序索引，创建索引的语句如下。

```
create index idx1 on CD_U_MC_MODEL_REMINDER_ORGCUST(procod,assign_orgno,model_id,reach_date desc);
```



创建索引后，该 SQL 语句的执行计划如下所示。

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Times
0	SELECT STATEMENT		50	131K		150 00:00:01
*1	VIEW		50	131K		150 00:00:01
*2	COUNT STOPKEY					
3	VIEW		50	131K		150 00:00:01
4	NESTED LOOP		50	10600		149 00:00:01
*5	NESTED LOOP		52	10600		149 00:00:01
*6	TABLE ACCESS BY INDEX ROWID	CD_U_MC_MODEL_REMINDER_ORGCUST	222	40781		50 00:00:01
*7	INDEX RANGE SCAN	IDX1	51			5 00:00:01
*8	INDEX RANGE SCAN	R_MODEL_INDEX	1			3 00:00:01
*9	TABLE ACCESS BY INDEX ROWID	CD_U_MC_MODEL_REMINDER	1	26		1 00:00:01

通过执行计划可以发现，排序操作已被消除，优化器选择了嵌套循环关联，并以排序序列所属的 CD_U_MC_MODEL_REMINDER_ORGCUST 表作为嵌套循环关联的驱动表。

实际执行该 SQL 语句，查询用时降至 0.6 秒左右。

继续对该查询进行优化，通过进一步分析

CD_U_MC_MODEL_REMINDER_ORGCUST 表的数据分布，发现该表的记录按 procode 列的数据分布，整个表中 procode 列的值约有 30 个，每个值对应的记录数在 100 万-300 万条左右，查询执行时对 CD_U_MC_MODEL_REMINDER_ORGCUST 表按 procode 的值进行了过滤。因此，我们可以将 CD_U_MC_MODEL_REMINDER_ORGCUST 表按 procode 的值进行分区，创建非 HASH/LIST 分区。此外，每次查询时 procode 只指定一个值，不存在跨分区查询，所以，为 procode 列创建本地索引即可。

经过以上第二次调优后，该 SQL 语句的执行计划如下所示。

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Times	Pstart	Pstop
0	SELECT STATEMENT		50	131K		120 00:00:01		
*1	VIEW		50	131K		120 00:00:01		
*2	COUNT STOPKEY							
3	VIEW		50	131K		120 00:00:01		
4	NESTED LOOP		50	10600		120 00:00:01		
5	NESTED LOOP		52	10985		6 00:00:01		
6	PARTITION LIST SINGLE		292K	50M		6 00:00:01	22	22
7	TABLE ACCESS BY LOCAL INDEX ROWID	CD_U_MC_MODEL_REMINDER_ORGCUST	292K	50M		5 00:00:01	22	22
*8	INDEX RANGE SCAN	INDEX_ORGCUST3	51			5 00:00:01	22	22
*9	INDEX RANGE SCAN	R_MODEL_INDEX	1			3 00:00:01		
*10	TABLE ACCESS BY INDEX ROWID	CD_U_MC_MODEL_REMINDER	1	26		1 00:00:01		

Predicate Information (identified by operation id):

- 1 - filter("RNUM">=1)
- 2 - filter(ROWNUM<=10)
- 8 - access("A"."ASSIGN_ORGNO"='992400000000' AND "A"."MODEL_ID"='xnC8100000000810')
- 9 - access("B"."MODEL_ID"='xnC8100000000810')
- 10 - filter(("B"."ROLE_ID_ID" IS NULL OR "B"."ROLE_ID_ID"='BASEPFOO15') AND "B"."STATUS"='00')

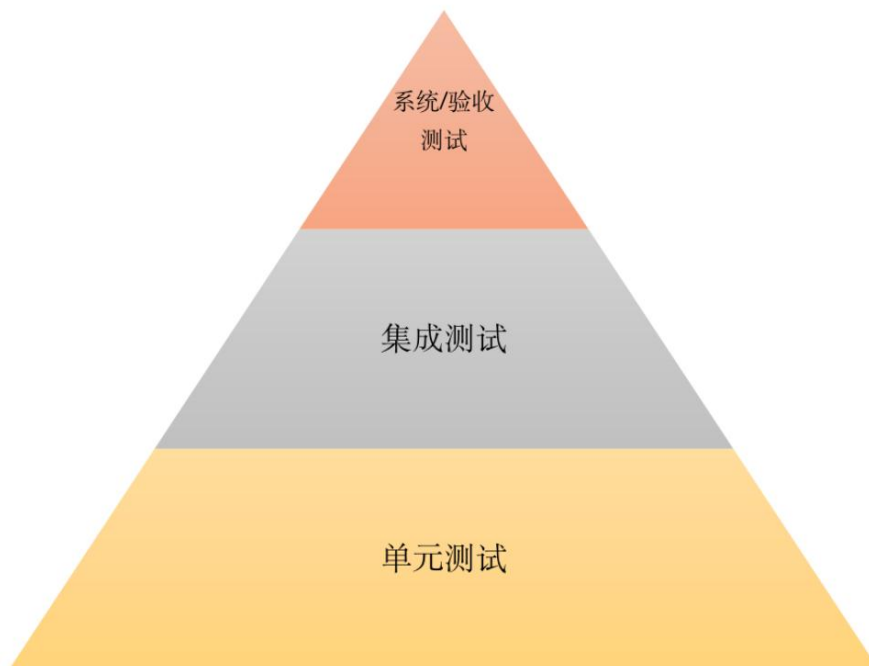
通过以上的执行计划可以发现，对 CD_U_MC_MODEL_REMINDER_ORGCUST 表的访问只扫描了一个分区，该 SQL 语句的执行时间进一步降低至 10 毫秒左右。



如何更高效地开展单元测试

◆作者：有房车的直男

很多测试人员对单元测试其实也就停留在面试环节，也就是面试官问到有几个测试阶段会，测试猿一上来会提到单元测试，而且很多人会片面的把单元测试归到白盒测试，然而，需要指出的是，单元测试也并非完全排斥黑盒测试的思想。在某些情况下，对于一些相对独立、功能明确的单元，也可以从外部的功能角度进行类似于黑盒测试的设计，这点可能也会颠覆很多测试猿的认知。随着中国软件研发越来越规范，单元测试是一项不可或缺的重要环节。它不仅能够确保代码的正确性和稳定性，还能在一定程度上降低软件维护成本，提高开发效率。所以很多公司也开始要求测试人员对软件进行单元测试，而不再单单让开发自测。本文将围绕单元测试的五个关键问题展开讨论，并结合具体项目的代码实例，探讨如何更高效地开展单元测试。



1. 什么是单元测试？

单元测试是指对软件中的最小可测试单元进行验证的过程。通常，这个单元指的是一个函数或一个方法。单元测试的目标是验证这个单元的正确性，即它是否按照预期执行。

示例代码

假设我们有一个简单的函数，用于计算两个数的和：

```
1 def add(a, b):
2     return a + b
```

我们可以为这个函数编写一个简单的单元测试：

```
1 import unittest
2
3 class TestAddFunction(unittest.TestCase):
4     def test_add(self):
5         self.assertEqual(add(1, 2), 3)
6         self.assertEqual(add(-1, 1), 0)
7         self.assertEqual(add(0, 0), 0)
8
9 if __name__ == '__main__':
10     unittest.main()
```

在这个例子中，我们使用了`unittest`模块，这是 Python 标准库中的一个单元测试框架。通过编写这样的测试用例，我们可以验证`add`函数在不同输入下的输出是否正确。

2. 为什么要进行单元测试？

进行单元测试有多个好处，主要包括以下几点：

提高代码质量

通过编写单元测试，可以确保代码在开发的早期阶段就能被充分验证。这有助于发现并修复潜在的缺陷，减少后期出现的问题，从而提高代码的整体质量。



便于重构

在软件开发过程中，重构是不可避免的。通过单元测试，可以在重构后快速验证代码的正确性，确保新代码不会引入新的错误。

降低维护成本

单元测试有助于捕获回归错误，即在修改代码时引入的旧错误。通过自动化测试，可以在修改代码后快速发现并修复这些错误，从而降低维护成本。

提高开发效率

尽管编写单元测试需要花费一定的时间，但它可以帮助开发人员更快地发现和解决问题，从长远来看可以提高开发效率。

3. 如何编写高质量的单元测试？

编写高质量的单元测试需要注意以下几点：

独立性

每个单元测试应该是独立的，不能依赖于其他测试的结果。这样可以确保测试的可重复性，即每次运行测试时的结果是相同的。

易读性

单元测试应该易于理解，这样不仅有助于开发人员在编写和维护测试时的效率，也有助于其他人理解代码的功能和预期行为。

完整性

单元测试应该覆盖代码的所有功能和边界情况。只有这样才能确保代码在各种情况下的正确性。

简洁性

单元测试应该尽量简洁，避免包含过多的逻辑。这有助于减少测试本身的错误，提高测试的可靠性。



示例代码

下面是一个更复杂的例子，展示了如何编写高质量的单元测试：

```

1  def is_prime(n):
2      """判断一个数是否为质数"""
3      if n <= 1:
4          return False
5      for i in range(2, int(n ** 0.5) + 1):
6          if n % i == 0:
7              return False
8      return True
9
10 class TestIsPrimeFunction(unittest.TestCase):
11     def test_is_prime(self):
12         self.assertTrue(is_prime(2))
13         self.assertTrue(is_prime(3))
14         self.assertFalse(is_prime(4))
15         self.assertTrue(is_prime(5))
16         self.assertFalse(is_prime(-1))
17         self.assertFalse(is_prime(0))
18         self.assertFalse(is_prime(1))
19
20 if __name__ == '__main__':
21     unittest.main()

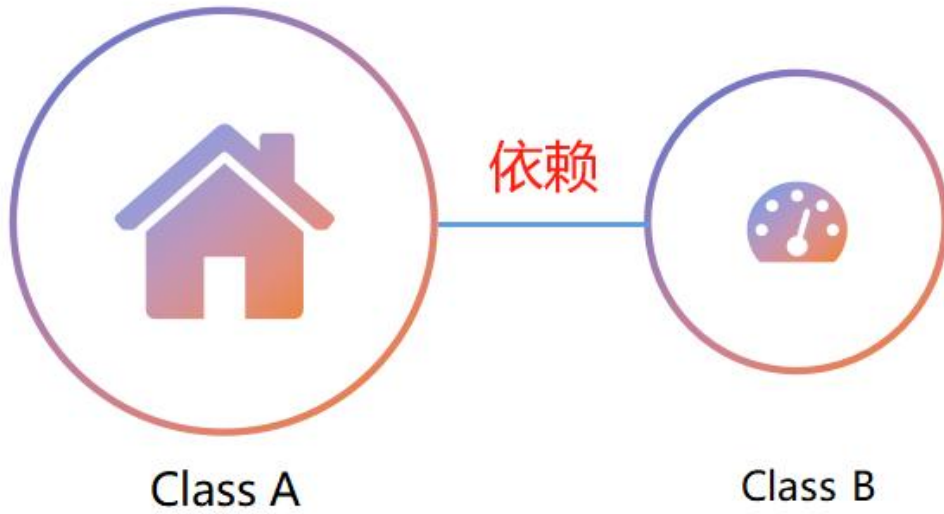
```

在这个例子中我们为`is_prime`函数编写了多个测试用例，覆盖了不同的输入情况，包括质数和非质数的情况。

4. 如何处理依赖和外部资源？

相比集成测试或系统测试，单元测试的一个重要特点是非常依赖 Mock。所谓 Mock，就是用模拟对象替换被测代码的依赖，它本质上是测试环境的一部分。例如，一个函数可能依赖于数据库或外部 API。为了确保单元测试的独立性和可重复性，我们需要对这些依赖进行隔离。





使用 Mock 对象

Mock 对象可以用来模拟依赖和外部资源的行为，从而使单元测试可以在不依赖真实资源的情况下进行。

示例代码

假设我们有一个函数，需要从外部 API 获取数据：

```
1 import requests
2
3 1 usage
4 def fetch_data(url):
5     response = requests.get(url)
6     return response.json()
7
8 class TestFetchDataFunction(unittest.TestCase):
9     @patch('requests.get')
10    def test_fetch_data(self, mock_get):
11        mock_response = Mock()
12        mock_response.json.return_value = {'key': 'value'}
13        mock_get.return_value = mock_response
14
15        result = fetch_data('http://example.com/api')
16        self.assertEqual(result, {'key': 'value'})
17
18 if __name__ == '__main__':
19     unittest.main()
```



在这个例子中，我们使用了`unittest.mock`模块中的`patch`装饰器来替换`requests.get`函数，并使用 Mock 对象来模拟 API 的响应。这样，我们可以在不依赖真实 API 的情况下测试`fetch_data`函数。

5. 如何管理和维护单元测试？

管理和维护单元测试是确保测试长期有效的关键。以下是一些管理和维护单元测试的最佳实践：

定期运行测试

定期运行单元测试可以确保代码在修改后仍然是正确的。可以使用持续集成工具（如 Jenkins、Travis CI 等）来自动化运行测试。

更新测试

在代码修改或添加新功能时，应该相应地更新或添加单元测试，以确保新代码的正确性。

清理无效测试

随着项目的发展，有些测试可能变得无效或过时。定期清理这些测试，可以保持测试的有效性和可靠性。

文档化测试

对测试进行文档化，可以帮助其他开发人员理解测试的目的和用法。这有助于提高团队的协作效率。

示例代码

假设我们有一个计算器类，并且对其进行了多次修改。我们需要确保每次修改后相应的单元测试也得到了更新：



```

1 class Calculator:
2     1 usage
3     def add(self, a, b):
4         return a + b
5
6     1 usage
7     def subtract(self, a, b):
8         return a - b
9
10    1 usage
11    def multiply(self, a, b):
12        return a * b
13
14    2 usages
15    def divide(self, a, b):
16        if b == 0:
17            raise ValueError("Cannot divide by zero")
18        return a / b
19
20 class TestCalculator(unittest.TestCase):
21     def setUp(self):
22         self.calculator = Calculator()
23
24     def test_add(self):
25         self.assertEqual(self.calculator.add( a: 1, b: 2), 3)
26
27     def test_subtract(self):
28         self.assertEqual(self.calculator.subtract( a: 2, b: 1), 1)
29
30     def test_multiply(self):
31         self.assertEqual(self.calculator.multiply( a: 2, b: 3), 6)
32
33     def test_divide(self):
34         self.assertEqual(self.calculator.divide( a: 6, b: 3), 2)
35         with self.assertRaises(ValueError):
36             self.calculator.divide( a: 1, b: 0)
37
38 if __name__ == '__main__':
39     unittest.main()

```

在这个例子中，我们在`setUp`方法中初始化了`Calculator`对象，并为每个方法编写了相应的测试用例。每次修改`Calculator`类的代码时，我们都需要相应地更新这些测试用例。

总结

单元测试是确保软件质量的重要手段。通过回答和解决以上五个关键问题，我们可以更高效地开展单元测试：



1. 理解单元测试的定义和目的，确保测试覆盖代码的核心功能和边界情况。
2. 意识到单元测试的好处，如提高代码质量、便于重构、降低维护成本和提高开发效率。
3. 编写高质量的单元测试，确保测试的独立性、易读性、完整性和简洁性。
4. 处理依赖和外部资源，通过使用 **Mock** 对象等技术手段来隔离依赖，确保测试的独立性和可重复性。
5. 管理和维护单元测试，通过定期运行测试、更新测试、清理无效测试和文档化测试来保持测试的有效性。

通过这些实践，开发团队可以更好地利用单元测试，减少后期可能会暴露出的 **bug**，提高软件的质量和稳定性，从而为用户提供更优质的产品。

----- END -----

